

# **Development and analysis of a distributed control strategy for power electronic converters**

Jacques A. du Toit

Dissertation presented in fulfilment of the requirements for the degree of Doctor of  
Philosophy in Engineering at the University of Stellenbosch

Supervisor: Dr H.J. Beukes



## **Declaration**

I, the undersigned, hereby declare that the work contained in the dissertation is my own original work, unless otherwise stated, and has not previously, in its entirety or in part, been submitted at any university for a degree.

J.A. du Toit

8 February 2002



## Opsomming

Die proefskrif ondersoek 'n alternatiewe manier om drywings omsetters te beheer. Huidiglik word die meerderheid van omsetters beheer vanaf 'n sentrale beheereenheid, wat dan stelsel data versamel vanaf een of meer meetstelsels. Soos die drywingsvermoë van die omsetters toeneem, raak spannings isolasie en die aantal beheerseine 'n probleem, wat nadelig is vir die betroubaarheid en vervaardigbaarheid van die stelsel. As 'n alternatief, kan 'n aantal kleiner beheereenhede en meetstelsels gebruik word. Die beheerders kommunikeer met mekaar, sowel as die hoof stelselbeheerder deur middel van 'n optiese vesel netwerk, verbind in 'n ring struktuur. Die proefskrif ondersoek die ontwerp van die beheerder en die toepassing van gedesentraliseerde beheer in 'n aantal nuwe omsetter topologië. Die studie toon dat sentrale beheer problematies kan raak soos die omsetters meer kompleks raak. Die studie bewys dat 'n stelsel suksesvol bedryf kan word deur gebruik te maak van gedesentraliseerde beheer deur dit in 'n praktiese opstelling toe te pas.

## Summary

The dissertation presents an alternative approach to the control of power electronic converters. The conventional approach is to use a centralized controller with one or more measurement systems providing feedback. As converters become larger, in both power rating and complexity, a number of drawbacks to this approach emerge. The number of physical data paths increases and voltage isolation becomes a problem. This has an adverse effect on the manufacturability as well as the reliable operation of the system as a whole. An alternative is to use a distributed control approach, where a number of smaller integrated control and measurement units are used. These units communicate with the central controller via a serial daisy-chain communications link. The dissertation investigates the design of such a controller as well as the application of distributed control in a number of emerging converter topologies. It was shown that centralized control has its limitations in modern power electronics in terms of reliability, maintainability and manufacturability. A feasible distributed control strategy was proposed and implemented and the operation was verified in an experimental converter system.

## **Acknowledgements**

I would like to thank the following people and institutions:

Firstly, I would like to thank my project leader, Johan Beukes, for his guidance and support.

Dr Johan Enslin for initiating this project and helping me to get exposure to the wide field of study required for this project.

Pietro Petzer and all the people working in the electrical workshop for their support on the more practical aspects of the project.

All my colleagues past and present for your help and support.

Eskom, the Foundation for Research and Development and the University of Stellenbosch for their financial support.



## Table of Contents

<b>CHAPTER 1 INTRODUCTION .....</b>	<b>2</b>
1.1 INTRODUCTION .....	2
1.2 DISTRIBUTED CONTROL APPLIED TO POWER ELECTRONICS .....	2
1.3 STANDARDIZATION .....	4
1.4 REQUIRED BACKGROUND STUDY .....	4
1.5 SUMMARY OF CONTRIBUTIONS .....	5
<b>CHAPTER 2 BACKGROUND OF POWER CONVERTER TOPOLOGIES AND CONTROL...8</b>	
2.1 INVESTIGATION OF POWER CONVERTER TOPOLOGIES .....	8
2.1.1 <i>Introduction</i> .....	8
2.1.2 <i>Basic topologies</i> .....	10
2.1.3 <i>Multi-device topologies</i> .....	12
2.2 BASIC BUILDING BLOCKS .....	19
2.3 SUMMARY .....	21
<b>CHAPTER 3 ANALYSIS OF THE DISTRIBUTED CONTROL STRATEGY .....</b>	<b>23</b>
3.1 INTRODUCTION .....	23
3.2 TRADITIONAL CONTROL METHODS .....	23
3.2.1 <i>Dynamic Power-Quality Compensator (DPQC)</i> .....	23
3.2.2 <i>Series stacked converter utilizing PEC31 controller</i> .....	24
3.3 DISTRIBUTED CONTROL STRATEGY .....	24
3.3.1 <i>Connecting the blocks</i> .....	24
3.4 CONFIGURATION OPTIONS FOR MODULE CONTROLLER .....	28
3.4.1 <i>Current regulator configuration</i> .....	28
3.4.2 <i>Voltage regulator configuration</i> .....	29
3.4.3 <i>DC-bus over-voltage protection</i> .....	30
3.4.4 <i>Capacitor center tap regulator</i> .....	31
3.5 PROPOSED MODULE CONTROLLER .....	32
3.5.1 <i>Requirements</i> .....	32
3.5.2 <i>Functional block diagram of proposed controller</i> .....	32

3.6 MAIN OUTER-LOOP CONTROLLER .....	33
3.7 SYSTEM CONFIGURATIONS .....	33
3.7.1 Shunt UPS using stacked converters .....	34
3.7.2 Series voltage regulator using a cascade multilevel converter.....	37
3.8 ANALYSIS .....	40
3.8.1 Manufacturability.....	40
3.8.2 Reliability.....	40
3.8.3 Controllability .....	41
3.9 SUMMARY .....	42
<b>CHAPTER 4 INTERFACE REQUIREMENTS OF THE MODULE CONTROLLER .....</b>	<b>45</b>
4.1 INTRODUCTION .....	45
4.2 INTERFACES TO THE MODULE CONTROLLER.....	45
4.3 POWER SWITCH TERMINALS.....	45
4.4 POWER SUPPLY.....	46
4.5 COMMUNICATIONS INTERFACE.....	48
4.5.1 Serial communications background.....	49
4.5.2 Standard communications interfaces.....	54
4.5.3 Synchronization and data transparency .....	55
4.5.4 Requirements of the communications link.....	57
4.6 SUMMARY .....	58
<b>CHAPTER 5 DESIGN OF THE MODULE CONTROLLER .....</b>	<b>61</b>
5.1 INTRODUCTION .....	61
5.2 DETAILED BLOCK DIAGRAM OF MODULE CONTROLLER .....	61
5.3 CONTROLLER AND GATE-DRIVE POWER SUPPLIES .....	62
5.4 COMMUNICATIONS INTERFACE.....	64
5.4.1 Development of communications link .....	64
5.5 MAIN CONTROL BOARD.....	69
5.5.1 Digital section.....	69
5.5.2 Analog section .....	73
5.6 MEASUREMENT BOARD .....	75



5.6.1 Voltage measurement.....	76
5.6.2 Current measurement.....	77
5.7 PROTECTION OF POWER MODULE AND CONTROLLER.....	77
5.8 MODULE SYNCHRONIZATION.....	79
5.9 CONTROLLER SOFTWARE.....	80
5.9.1 Module controller software timing.....	81
5.9.2 Operational states of the controller.....	82
5.9.3 Program flow.....	83
5.10 HOST INTERFACE SOFTWARE.....	86
5.10.1 Data flow through layers.....	86
5.11 SUMMARY.....	90
<b>CHAPTER 6 EVALUATION OF THE DISTRIBUTED CONTROLLER.....</b>	<b>92</b>
6.1 INTRODUCTION.....	92
6.2 EVALUATION OF MODULE CONTROLLER SUBSYSTEMS.....	93
6.2.1 Optic fiber quantization circuit.....	93
6.2.2 Distributed AC power supply.....	93
6.2.3 Protection.....	97
6.2.4 Data acquisition.....	98
6.2.5 Synchronization.....	100
6.3 EVALUATION OF MODULE WITH STANDARD CONFIGURATIONS.....	102
6.3.1 Current regulator.....	102
6.3.2 Voltage regulator.....	103
6.4 CONVERTER TOPOLOGIES.....	104
6.4.1 Three-phase with coupled outputs.....	104
6.4.2 Series stacked multilevel.....	106
6.5 SUMMARY.....	108
<b>CHAPTER 7 CONCLUSIONS.....</b>	<b>110</b>
7.1 CONVERTER TOPOLOGIES AND CONTROL.....	110
7.2 DISTRIBUTED CONTROL.....	110
7.3 DEVELOPMENT OF THE POWER BLOCK.....	111

7.4 EVALUATION .....111

7.5 COMMENTS AND FUTURE WORK .....112

**CHAPTER 8 REFERENCES .....114**

**A. MODCON SCHEMATICS .....120**

**B. PLD FIRMWARE SOURCE.....135**

**C. DSP SOFTWARE SOURCE.....151**

**D. PC SOFTWARE SOURCE .....194**

## List of Figures

Figure 1-1 – Block diagram of power block .....	3
Figure 1-2 – Main fields of study .....	4
Figure 2-1 – Basic converter building blocks (half-bridge) .....	8
Figure 2-2 – Half-bridge control signals .....	9
Figure 2-3 – Basic converter building blocks (full-bridge) .....	9
Figure 2-4 – 3 $\phi$ Inverter with coupled outputs .....	10
Figure 2-5 – Three-phase, four-wire topology .....	11
Figure 2-6 – 4 $\phi$ Converter with switched neutral .....	11
Figure 2-7 – Four-phase converter with clamped capacitor center tap .....	12
Figure 2-8 – Direct parallel connected devices (single phase) .....	13
Figure 2-9 – Direct series connected devices (single phase) .....	14
Figure 2-10 – Parallel connected phase-arms (single phase) .....	15
Figure 2-11 – Diode-clamped multilevel (single phase) .....	16
Figure 2-12 – Flying capacitor multicell converter (cell representation) .....	16
Figure 2-13 – Flying capacitor multicell converter (topology representation) .....	17
Figure 2-14 – Series connected converters (single phase) .....	17
Figure 2-15 – Cascade multilevel (single phase) .....	18
Figure 2-16 – Basic converter blocks (half-bridge) .....	19
Figure 2-17 – Basic converter blocks (full-bridge) .....	19
Figure 2-18 – Basic converter blocks (imbricated-cell) .....	20
Figure 3-1 – DPQC topology .....	24
Figure 3-2 – Module interconnect options .....	25
Figure 3-3 – Star configuration .....	26
Figure 3-4 – Hub configuration .....	26
Figure 3-5 – Ring configuration .....	27
Figure 3-6 – Current regulator configuration .....	28
Figure 3-7 – Voltage regulator configuration .....	29
Figure 3-8 – DC-dump configuration .....	30



Figure 3-9 – Gating signals for DC-bus over-voltage configuration .....	30
Figure 3-10 – Capacitor center tap regulator.....	31
Figure 3-11 – Functional block diagram of proposed module controller .....	33
Figure 3-12 – Single phase of shunt UPS using stacked converters (centralized control) .....	35
Figure 3-13 – Single phase of shunt UPS using stacked converters (distributed control) .....	36
Figure 3-14 – Series dip compensator using cascaded converters (centralized control) .....	38
Figure 3-15 – Series dip compensator using cascaded converters (distributed control) .....	39
Figure 4-1 – Module controller interfaces .....	45
Figure 4-2 – Power supplies for controller and gate-drives (DC distribution).....	47
Figure 4-3 – Power supplies for controller and gate-drives (AC distribution).....	47
Figure 4-4 – ISO reference model for communications subsystems.....	48
Figure 5-1 – Detailed block diagram of module controller .....	61
Figure 5-2 – Interconnection hierarchy .....	62
Figure 5-3 – Transformer layout .....	63
Figure 5-4 – Photograph of the gate interface and power supply .....	63
Figure 5-5 – Interconnecting the module controllers .....	65
Figure 5-6 – RS232 optic fiber link .....	66
Figure 5-7 – RS232 packet formats.....	66
Figure 5-8 – Description of packet fields .....	67
Figure 5-9 – Digital controller block diagram .....	69
Figure 5-10 – Block diagram of EPLD firmware (slave) .....	70
Figure 5-11 – Block diagram of EPLD firmware (master).....	71
Figure 5-12 – Block diagram of controller analog section .....	73
Figure 5-13 – Top view of module controller main board .....	74
Figure 5-14 – Bottom view of module controller main board .....	75
Figure 5-15 – Block diagram of measurement unit.....	75
Figure 5-16 – Photograph of measurement unit.....	76
Figure 5-17 – Ground reference points for controller .....	76
Figure 5-18 – Internal timing of the controller.....	81
Figure 5-19 – Controller stage timing .....	82



Figure 5-20 – Flow-chart of embedded code (main and interrupt entry/exit) .....	84
Figure 5-21 – Flow-chart of embedded code (stage specific) .....	85
Figure 5-22 – Data flow through the host software .....	86
Figure 5-23 – Final message structure.....	89
Figure 5-24 – Server status window .....	89
Figure 5-25 – Client application window.....	90
Figure 6-1 – Photograph showing converter stack.....	92
Figure 6-2 – Operation of quantization circuit.....	93
Figure 6-3 – Photograph of the distributed AC supply .....	94
Figure 6-4 – Switching waveform under full load .....	95
Figure 6-5 – Switching transient under light load (single device).....	95
Figure 6-6 – Switching transient under full load .....	96
Figure 6-7 – Output of fiber optic quantizer .....	96
Figure 6-8 – Reaction time after a trip condition.....	97
Figure 6-9 – Current after filter capacitors.....	98
Figure 6-10 – Current before filter capacitors.....	99
Figure 6-11 – Sampling instants of data acquisition unit.....	99
Figure 6-12 – Current sampled by module controller .....	100
Figure 6-13 – Carrier sync pulse received by slave .....	101
Figure 6-14 – Synchronization of modules using PLL.....	101
Figure 6-15 – Test setup for current regulator configuration .....	102
Figure 6-16 – Current regulation using module controller .....	103
Figure 6-17 – Current measured by module controller (all 4 sampling instants).....	103
Figure 6-18 – Test setup for voltage regulator configuration.....	104
Figure 6-19 – Voltage regulation with module controller .....	104
Figure 6-20 – Three-phase with coupled outputs test setup .....	105
Figure 6-21 – Three-phase converter test .....	105
Figure 6-22 – Series stacked multilevel converter test setup.....	106
Figure 6-23 – Operation of series stacked converter .....	107
Figure 6-24 – Natural balancing of three-level stacked converter.....	107

Figure 6-25 – Forced balancing of three-level stacked converter .....	108
Figure 8-1 – Module controller (DSP) .....	121
Figure 8-2 – Module controller (Power) .....	122
Figure 8-3 – Module controller (EPLDs) .....	123
Figure 8-4 – Module controller (Fiber optic) .....	124
Figure 8-5 – Module controller (Protection) .....	125
Figure 8-6 – Module controller (Supply measurements) .....	126
Figure 8-7 – Module controller (Voltage measurements) .....	127
Figure 8-8 – Module controller (Current measurement) .....	128
Figure 8-9 – Power-supply (Full-bridge) .....	129
Figure 8-10 – Power-supply (Gate pulse) .....	130
Figure 8-11 – Gate interface (Isolation) .....	131
Figure 8-12 – Gate interface (Power) .....	132
Figure 8-13 – Gate interface (Transformers) .....	133

## List of Tables

Table 3-1 – Current regulator requirements .....	29
Table 3-2 – Voltage regulator requirements .....	29
Table 3-3 – DC-bus over-voltage protection requirements .....	31
Table 3-4 – Capacitor center tap regulator requirements.....	32
Table 3-5 – Required functions of proposed controller .....	32
Table 3-6 – Summary of connections for series stacked converter .....	37
Table 3-7 – Summary of connections for cascade converter .....	39
Table 5-1 – Command definitions of module controller .....	67
Table 5-2 – Predefined data formats .....	69
Table 5-3 – Operational states of the controller .....	83



## List of Symbols and Abbreviations

### Power electronics, power systems

3 $\phi$	Three-phase
DC	Direct current
AC	Alternating current
IGBT	Insulated gate bipolar transistor
GTO	Gate turn-off thyristor
APF	Active Power Filter
VSI	Voltage Source Inverter

### Digital electronics, software

DSP	Digital signal processor
SRAM	Static RAM, Static Random Access Memory
bps	Bits per second
Bps	Bytes per second
Bd	Baud (symbols per second)
ADC	Analog to digital converter
DAC	Digital to analog converter
EPLD	Erasable programmable logic device
FPGA	Field programmable gate array
MIPS	Million instructions per second

### Miscellaneous

ISO	International Standards Organization
OSI	Open Systems Interconnection

# Chapter 1

---

## Introduction

## Chapter 1 Introduction

### 1.1 Introduction

The focus of this dissertation is to investigate the applicability of a distributed control strategy to the field of power electronics, specifically the power electronic converter. The second important aspect is the development of a generic power block or cell. The design of the power block must not restrict the developer in the choice of converter topology or power rating (within certain limits, as set by the individual power block).

Distributed control is not a new concept and has been applied in a variety of engineering fields. There is a growing tendency to decentralize the control of medium and large-scale systems. Typical examples are servo control in robotics used for manufacturing and the automobile industry, where the so-called drive by wire concept is being investigated. For example, in a standard automobile vehicle, the brake pedal is connected to the wheel by using a hydraulic connection. With the drive by wire implementation, the position of the pedal is measured and an electrical control signal is sent to an actuator that activates the brake on the wheel. This implies that a high level of reliability is required, because the safety of the driver and passenger is of premium importance.

There are a number of advantages and disadvantages of a decentralized control strategy, which will be discussed in the following chapters. Performance criteria used during the evaluation include reliability, manufacturability and cost.

### 1.2 Distributed control applied to power electronics

The ever-increasing demand for better quality of supply leads to a bigger demand for high-performance converters as well as high-power converters. As the number of switching converters used by industry and the everyday consumer increases, the need for more stringent EMI (electro-magnetic interference) compatibility increases.

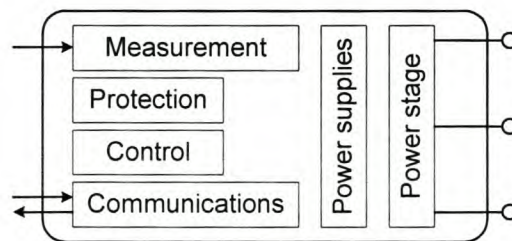
The high power rating required for large machinery and drives can be obtained in a number of ways. High-power drives (e.g. those used by the mining industry), typically use a cyclo-converter for the power stage. Other converters simply increase the DC-bus voltage or use parallel switches to increase the current rating. All these converters use brute force methods to increase the power rating and are typically associated with high levels of radiated and conducted EMI.



In order to lower the levels of EMI, the voltage and current transient must be suppressed. This can be done with resonant topologies like the ARCP (auxiliary resonant commutated pole converter) [23][24] and resonant DC and AC link converters [19]. A second approach is to use multilevel converters. These operate by breaking the output waveform into smaller steps by combining the outputs of a number of converter cells. Included in this category are the diode clamped multilevel converter, the flying capacitor multicell [10] converter and cascaded multicell [27] converter. A third approach is to use multiple converters and combine the outputs in series [26][44][43] or parallel. The converter topologies will be discussed in more detail in Chapter 2.

As the physical converter becomes more complex, the manufacturing and control become more difficult. Factors like cooling, voltage isolation and wiring layout become a major problem. Each device needs control and status feedback signals, as well as measurements and power. The integrity of the signalling is also more difficult to check due of the large number of states of these converters.

One possible solution would be to have a standardized power block with one input and one output signal used for control, status and measurements, a standard power supply input and connectors for the power switch. This power block would then be responsible for the control of the voltage or current output, protection and measurements. A block diagram for a generic power block is shown in Figure 1-1.



**Figure 1-1 – Block diagram of power block**

The self-controlled power block must be capable of functioning in a variety of modes. Depending on the application and specific placement in the converter, it must be able to operate as a voltage regulator, a current regulator or a DC-dump for regenerative loads. The various modes of operation are discussed in section 3.4 and evaluated in section 6.2. The solution presented in this dissertation is called the module controller. The design of the controller is discussed in Chapter 5.



Having a power block is necessary but not sufficient for distributed control. Further investigation must follow into the control aspects of a converter or power system based on the decentralized control approach. Certain converter topologies will be better suited for decentralized control than others will. This is discussed in section 6.4.

A region of control must also be established when evaluating distributed control. Certain topologies exhibit natural balancing of internal voltages and care must be taken not to influence the operation of the converter. For certain converters it might be better to expand the region of control to include a complete sub-converter and not base the control on an individual power module.

### 1.3 Standardization

It is important to standardize the interfaces to and from the power block. This will lead to lower manufacturing and assembly costs and ease the construction and upgrade of large-scale converters. A standard for the control and status interface will allow re-use of outer-loop controllers and allow for easier expansion of the converter.

### 1.4 Required background study

In order to investigate the distributed control of power electronics, a number of diverse fields had to be investigated and combined. The dissertation can be broken into four main fields of study and development, shown in Figure 1-2.

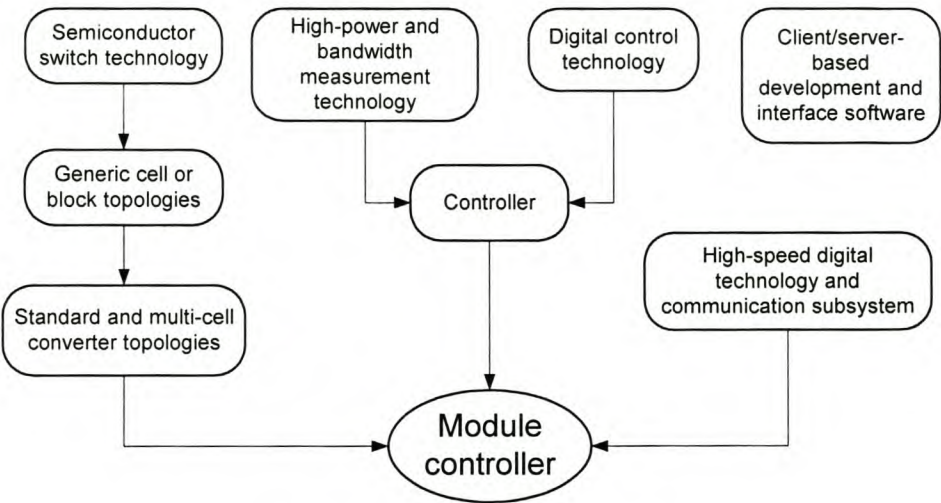


Figure 1-2 – Main fields of study

The first field concerns the power stage of the module controller. This includes the power switch technology, the power electronic converter and identifying basic building blocks for converters.

The second field covers the hardware needed for the control of the module controller. Included in this field are the measurement system, the digital controller and the power supply.

Distributed control implies that a number of separate control units exist, some or all of which must communicate with a master controller. This requires the use of a communications link. The power electronic application has very specific needs and this forms the third field of study required for the successful evaluation of the distributed control strategy.

The development of the required software forms the fourth and last field required. The software covers the embedded control of the module controller and the development software required to operate the evaluation system.

## **1.5 Summary of contributions**

This section gives a summary of the contributions to the fields of power electronics and control provided by this project. The main contributions are:

- (1) Background study of current and emerging converter topologies.
- (2) Identifying basic building blocks for the construction of converters.
- (3) Identifying the requirements for building a generic power block.
- (4) Background study on applicable serial communications protocols and transmission media for high-voltage applications.
- (5) Study of power supply topologies for high-voltage isolated circuits.
- (6) Establishing and evaluating configurations for the generic power block. Examples are current and voltage regulation.
- (7) The detailed design and evaluation of a power block, called the module controller.
- (8) Identifying the converter topologies that can benefit from distributed control and the implications for stability and control.



- (9) Identifying and implementing the protection required for the power block, with special attention given to the additional protection needed when using distributed control.
- (10) Two case studies of multilevel converter applications comparing the centralized control vs. distributed control implementations.
- (11) Evaluation of the module controller in a variety of converter configurations.
- (12) Identifying the software requirements for debugging and evaluating the inner working of the module controller and implementing this as part of the embedded controller software. This includes online inspection of values and waveform capture.
- (13) A client and server application running on one or more personal computers for communicating with the controllers of the distributed converter.

Each of the above contributions is discussed in the following chapters and summarized in the conclusions at the end of the dissertation.

# Chapter 2

---

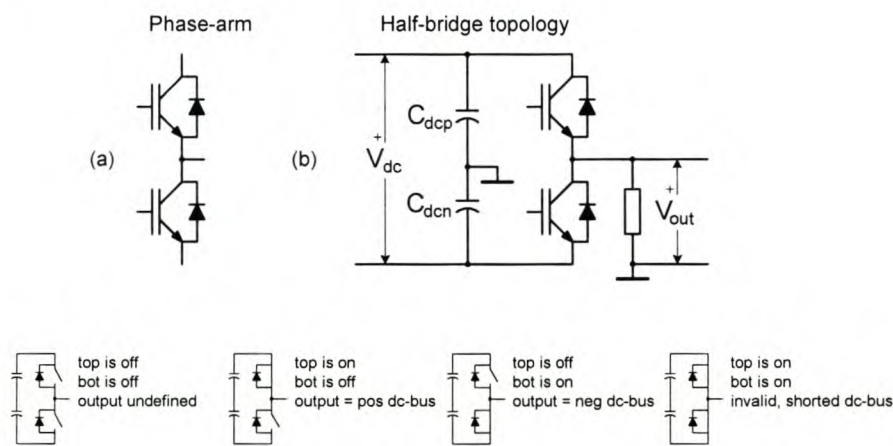
Background of power converter  
topologies and control

## Chapter 2 Background of power converter topologies and control

### 2.1 Investigation of power converter topologies

#### 2.1.1 Introduction

The basic building block of the power converters discussed in this dissertation is the phase-arm (also known as a phase-leg), shown in Figure 2-1(a). They can be used singly or in pairs, called half-bridge and full-bridge configurations, as shown in Figure 2-1(b) and Figure 2-3 respectively.



**Figure 2-1 – Basic converter building blocks (half-bridge)**

In order to fully understand the consequences of a distributed control strategy, the basic operation of each converter configuration was investigated. This is especially important with the multi-phase, multi-converter and multicell topologies.

The half-bridge circuit of Figure 2-1 has only two valid states, although four control signal combinations are possible (shown in Figure 2-1). The anti-parallel freewheeling diodes limit the possible output voltage levels to two, either  $+\frac{1}{2}V_{dc}$  and  $-\frac{1}{2}V_{dc}$ , or  $+V_{dc}$  and  $0V$ , depending on the ground reference point.

For a voltage source inverter (VSI), the DC-bus will be shorted if both switches are turned on; thus it is an illegal switching combination. If neither IGBT is explicitly turned on, the output voltage is undefined from a control perspective and the level at the output will depend on the direction of the output current flow. The only time this is allowed to happen is during the blanking time. The typical control signals fed to the half-bridge are shown in Figure 2-2.



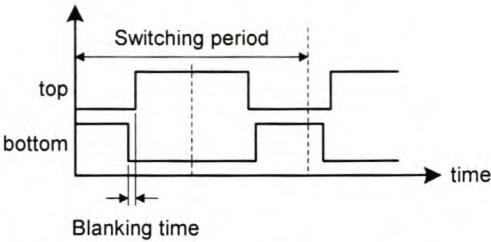


Figure 2-2 – Half-bridge control signals

The full-bridge configuration shown in Figure 2-3 is a combination of two half-bridge converters. The output is defined as the difference between the two half-bridge outputs. The switching combinations discussed for the half-bridge are applicable to the full-bridge converter, although the possible switching states have doubled.

There are four switching states (pp, nn, pn and np), but two of the states produce the same output voltage (called the zero states). The possible output voltages are  $+V_{dc}$ ,  $-V_{dc}$  and  $0V$ . Although the zero states produce the same output voltage, they should both be used to allow for a more even distribution of the conduction and switching losses in the switches. This is usually not a problem with more intelligent regulators (for example, the predictive current regulators) that make explicit use of the zero states. Current regulators like the standard hysteresis controller do not make explicit use of the zero states and produce only output levels of  $\pm V_{dc}$ .

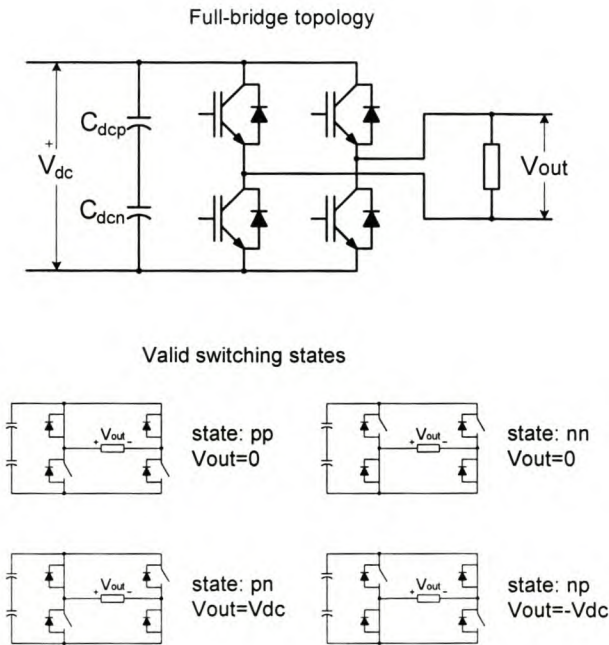


Figure 2-3 – Basic converter building blocks (full-bridge)



## 2.1.2 Basic topologies

The basic topologies discussed below are topologies where the output voltage or current is produced by a single half-bridge or full-bridge. The output of each phase has only two states and no multilevel outputs are possible. Outputs can either be coupled, as in the case of the standard three-phase converter, or de-coupled as with the three-phase, four-wire topology. The coupling, or cross-influencing of the outputs, has a fundamental influence on the controllability of the converter, especially the amount of system data needed for optimal control.

### 2.1.2.1 Three-phase converter with coupled outputs

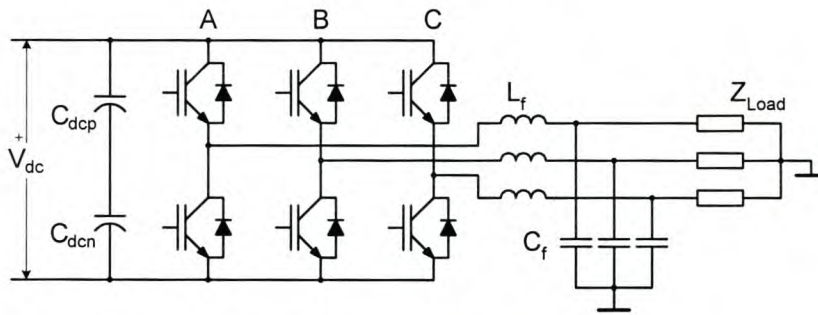


Figure 2-4 – 3 $\phi$  Inverter with coupled outputs

This is the most popular converter topology in use in the industry today. The outputs of the three phases are coupled; therefore, the output of one phase will directly influence the output of the other two phases. The star-point connection of the outputs (shown in Figure 2-4) will change depending on the state of the converter switches.

There are eight converter states, two of which are called the zero states (all top or all bottom switches on). Depending on the state, the forcing voltage can be either 0,  $\pm \frac{1}{3}V_{dc}$  or  $\pm \frac{2}{3}V_{dc}$ . The availability of the zero states (0 forcing voltage) allows for the reduction of the output current ripple. There is no path for zero sequence currents to flow (output currents must sum to zero).

Optimal control of this type of converter requires system-wide parameters (for example, load voltages, filter parameters and phase-arm currents) to be known. The modulation scheme takes into account all these parameters and predicts the output for the next switching cycle. Alternatives for optimal control are space-vector based hysteresis, etc.

### 2.1.2.2 Three-phase, four-wire topology

The star point of the connected outputs can be forced to a relatively fixed voltage by connecting the star-point to the DC-bus capacitor center point, as shown in Figure 2-5. The advantage of this scheme is that the outputs of the converter are de-coupled and optimal control can therefore be implemented with only local parameters required (local to the phase-arm, e.g. phase current and load voltage). The forcing voltage is either  $\frac{1}{2}V_{dc}$  or  $-\frac{1}{2}V_{dc}$ . There are no zero states and this has a negative influence on the ripple of the output current.

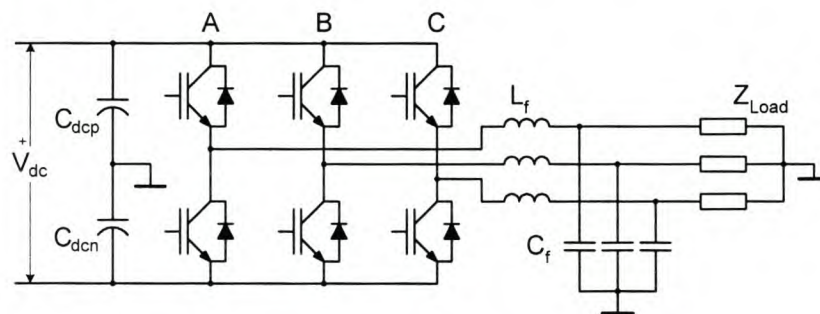


Figure 2-5 – Three-phase, four-wire topology

There is a path for zero sequence currents to flow, but this neutral current must be absorbed by the DC-bus capacitors. Care must therefore be taken to limit the voltage swing on the capacitors. For closed-loop current regulators, measurement offsets can easily introduce DC currents. It is therefore advisable to measure and control the DC-capacitor voltage, unless the output filter/load combination prevents the situation from occurring.

### 2.1.2.3 Four-phase converter with switched neutral

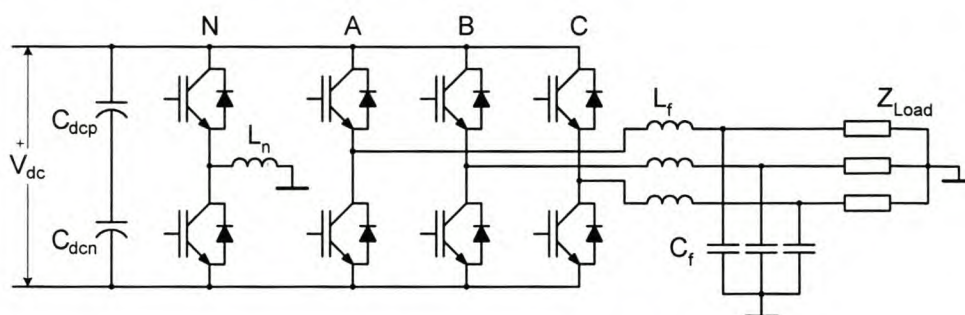


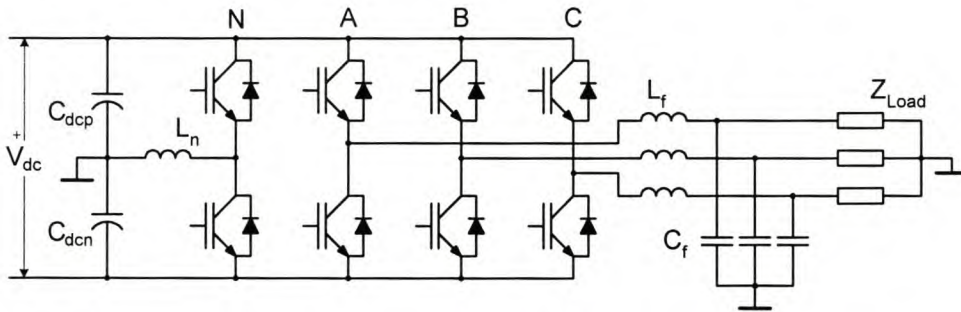
Figure 2-6 – 4φ Converter with switched neutral

An alternative to the four-wire topology described in section 2.1.2.2 is to connect the star-point of the outputs to the output of a fourth phase-arm. The star-point can therefore be forced to a specific (known) voltage.



There are two basic approaches to controlling the output of the fourth phase-arm. The first is to use a space vector controller operating in three dimensions (taking into account the zero sequence currents). The second method is to pulse width modulate the fourth phase-arm to emulate a fixed voltage, for example, the DC-bus capacitor center voltage.

#### 2.1.2.4 Four-phase converter with clamped capacitor center tap



**Figure 2-7 – Four-phase converter with clamped capacitor center tap**

The two topologies can also be combined to form the clamped capacitor center tap topology. The fourth phase-arm is used to regulate the capacitor voltages. The outputs of the three main phase-arms are completely de-coupled and each phase can be controlled independently. The converter can handle zero-sequence currents and the addition of the fourth phase-arm allows for smaller values of DC-bus capacitors.

#### 2.1.3 Multi-device topologies

This section describes a number of converter topologies where multiple devices are used to obtain a single output. These topologies are typically used where the current and/or voltage rating required is higher than a single power device can handle. For the discussion, it is assumed that the switches are IGBTs, mainly due to their wide application in the industry, although similar topologies are possible with other fully controlled semiconductor power devices.

The increased number of switching states opens up a completely new field of control possibilities, but with this comes a completely new set of problems. Voltage and current sharing between the switches must be taken into account, as well as the sharing of switching and conduction losses. Most of the converters have a tendency to balance naturally under steady state, but for applications requiring high dynamic



currents or voltages (for example, an active power filter, APF), care must be taken to prevent excessive unbalance and possible converter failure.

### 2.1.3.1 Direct parallel connected devices

The simplest way of increasing the current rating of a converter is to directly connect a number of switches in parallel, as shown in Figure 2-8. This configuration is also used in a number of standard IGBT modules, where multiple IGBTs are connected in parallel inside the module packaging [35][37].

The use of the module is completely transparent to the designer of the converter. Current sharing is not a major problem, because the IGBTs are closely matched. The positive temperature coefficient (i.e. the resistance increases with a rise in temperature) of the IGBTs will also help the current sharing if slight mismatches in the devices are present.

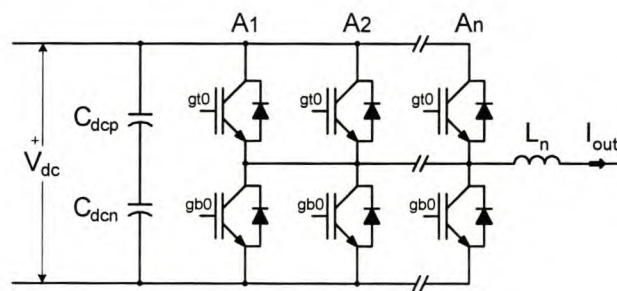


Figure 2-8 – Direct parallel connected devices (single phase)

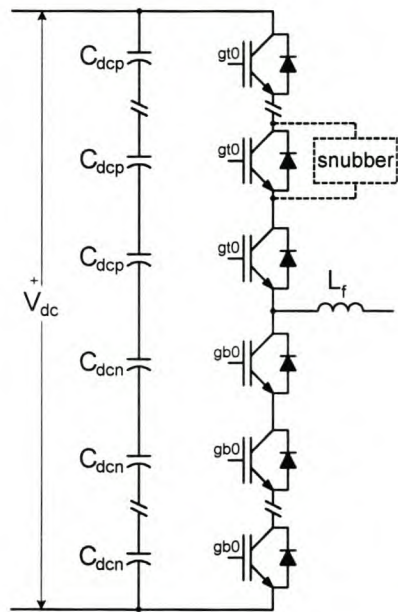
All the gate control signals are connected together in the top and bottom sets of IGBTs respectively and therefore no additional switching states are possible. The advantages of the configuration are the simple control of the converter and the low number of passive components. The disadvantage is the high current ripple, as all switches in the converter are switched simultaneously.

### 2.1.3.2 Direct series connected devices

If a higher DC-bus or output voltage is required, a number of switches can be used in series. This will increase the blocking capability of the converter. As in the case of the parallel devices, a number of configuration possibilities exist. The simplest configuration is to connect the devices directly in series, as shown in Figure 2-9.

In the ideal case all the switches in the top or bottom set would turn on and off at the exact same time and sharing of the bus voltage would result. In a practical setup this

is not true and extra components are needed to aid the voltage sharing. The addition of a snubber to each of the switches will slow the switching action and allow the voltages to share more evenly [44]. If faster switching times or more accurate voltage sharing is necessary, the IGBT collector-emitter ( $V_{CE}$ ) voltages can be measured and the gate-drive signals can be controlled actively. The IGBT voltages are thus shared by controlling the turn-off and turn-on times of the individual IGBTs.



**Figure 2-9 – Direct series connected devices (single phase)**

Although low-level control can be added, the converter still has only two states from a control perspective. Advantages of the topology are the higher DC-bus capability and the simple higher-level control of the converter. The disadvantages are the voltage-sharing problems and the high-voltage transients present at the output (output switches between  $+V_{dc}$  and 0).

### 2.1.3.3 Parallel connected phase-arms

An alternative configuration for increased the current rating is to connect the phase-arms in parallel through a number of filter inductors. The topology is shown in Figure 2-10, where the phase-arms are connected in parallel through the filter inductors  $L_{f1}$  to  $L_{fn}$ . The inductance values for the phase-arms may differ by design, although in a typical converter they will be the same, as this simplifies the control.



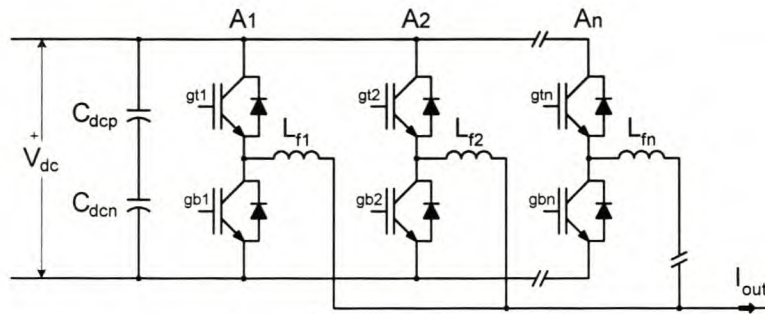


Figure 2-10 – Parallel connected phase-arms (single phase)

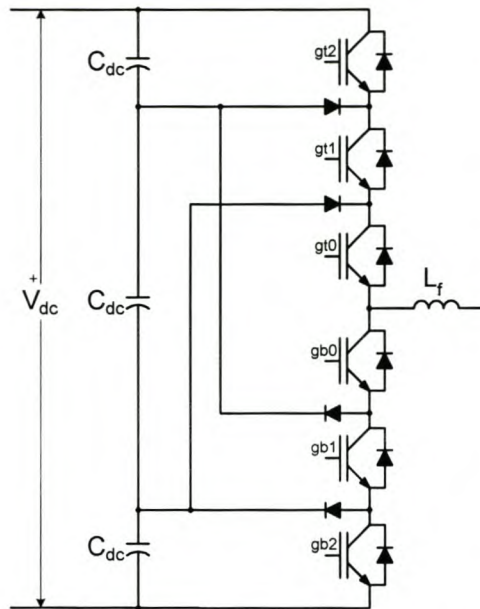
A number of approaches can be taken with the control of the converter. The simplest form of control would be to measure the combined output current ( $I_{out}$ ) and pass the same duty cycle to all the phase-arms. The control signals can be shifted relative to each other (called interleaved switching) and will result in a decrease of the output current ripple. Advantages of this configuration are that the output power can be increased arbitrarily by adding additional phase-arms and switching current harmonics can be decreased in both the output current and DC-bus capacitors. The main disadvantage is the addition of extra filter inductors.

#### 2.1.3.4 Diode-clamped multilevel

An alternative to the direct series connection of the switches is the diode-clamped multilevel converter shown in Figure 2-11. The levels of a multilevel converter are generally meant to indicate the number of voltage levels available at the output of the converter. A single phase-arm has two levels; either  $V_{dc}$  or 0, depending on the top or bottom switch conducting (refer to section 2.1.1 for more details).

The converter shown in Figure 2-11 has 4 levels ( $V_{dc}$ ,  $\frac{2}{3}V_{dc}$ ,  $\frac{1}{3}V_{dc}$  and 0). For each level added to the converter, two additional switches and two clamping diodes are needed. The required voltage blocking capability of the clamping diodes scale with the number of levels and typically a number of similar diodes are used in series.

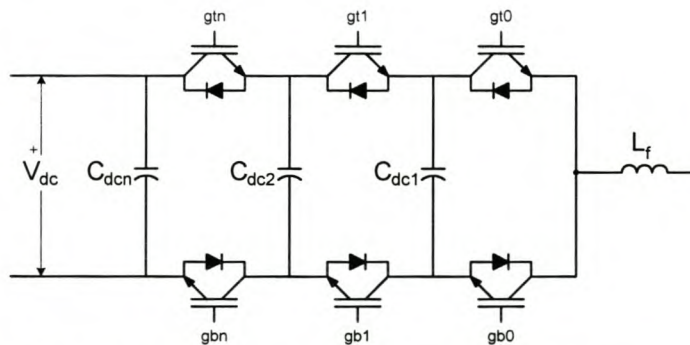
The number of output levels leads to finer control of the output voltage of the converter. For three-phase converters, the number of states increases and for low modulation indexes there are redundant voltage states. Redundant voltage states are those states where more than one switch combination can give the same output voltage. Utilization of these states is needed to allow active sharing of device usage (sharing of conduction and switching losses).



**Figure 2-11 – Diode-clamped multilevel (single phase)**

#### 2.1.3.5 Flying capacitor multicell converter

There are two distinct representations for this topology. The first representation, shown in Figure 2-13, shows the cell architecture of the topology. The alternative representation shows the series connection of the power switches, as shown in Figure 2-13. The topology is also known as the imbricated cell multilevel topology.



**Figure 2-12 – Flying capacitor multicell converter (cell representation)**

Each additional cell added to the converter enables the converter to handle a larger DC-bus voltage. The voltage and current rating of the power switches are identical, independent of the number of cells. The capacitance of the cell capacitors ( $C_{dc1}$  to  $C_{dcn}$ ) are the same, but the voltage rating of each additional cell must be able to handle the increased DC-bus voltage rating.



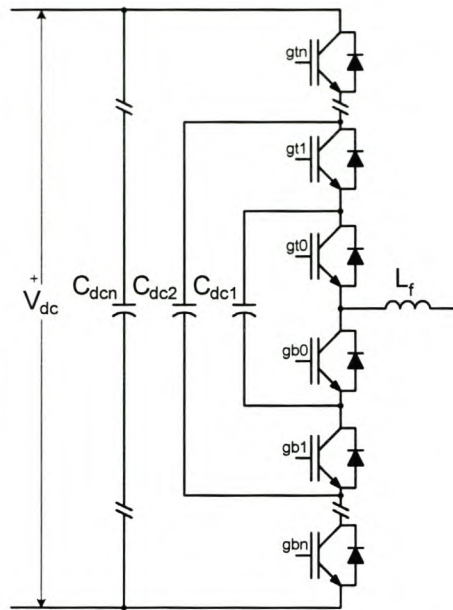


Figure 2-13 – Flying capacitor multicell converter (topology representation)

#### 2.1.3.6 Multi-converter, series stacked

A number of full-bridge converters can be connected in series, either by connecting the DC-buses or by connecting the output voltages in series. The first topology is called the series stacked multilevel converter [43] and the second topology is called the cascade multilevel converter [27], described in section 2.1.3.7.

The series stacked multilevel converter is shown in Figure 2-14. It consists of a number of similar full-bridge converter and filter combinations.

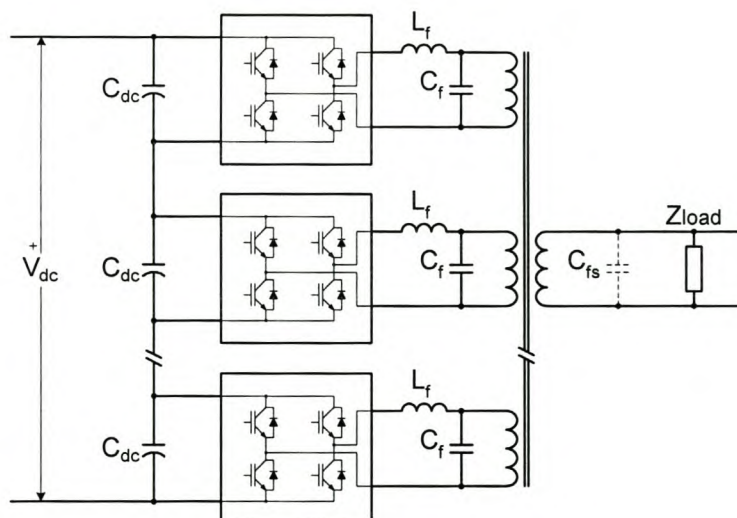


Figure 2-14 – Series connected converters (single phase)

The DC-buses of the converters are connected in series and the outputs are connected to matching primaries of the output transformer. The filter capacitors ( $C_f$ ) can also be removed and replaced with a single filter capacitor ( $C_{fs}$ ) on the output of the transformer. The disadvantage of this is that higher ripple currents will flow through the transformer windings, but the leakage inductance of the transformer can be used as part of the filter inductance.

The converter has excellent balancing characteristics under steady state conditions [43]. All the components of the converter, except for the transformer, are rated at a single level of the converter. This includes the capacitors and switches and no additional diodes are needed for clamping. The only disadvantage is the requirement of the specialized transformer, which has to be designed with a predetermined number of levels.

#### 2.1.3.7 Cascade multilevel

The cascade multilevel converter [28][30] is constructed by cascade connecting the outputs of a number of full-bridge converters together, as shown in Figure 2-15.

Each of the full-bridge converters, called a cell, has three states,  $+V_{dc}$ ,  $-V_{dc}$  and 0. Each cell can contribute a positive DC-bus voltage, a negative DC-bus voltage, or can be bypassed completely by switching the converter into one of the two zero states. For low injection voltages, there are a number of redundant voltage states and additional control must be applied to force equal device utilization [29].

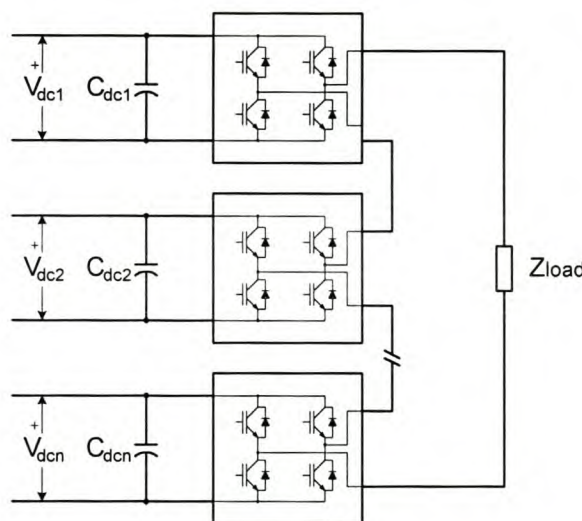


Figure 2-15 – Cascade multilevel (single phase)



The topology requires an isolated supply for each DC-bus. If the device is used purely for reactive power compensation, the isolated buses pose no problem. The energy source can simply be replaced by a capacitor. The source of energy for the bus can be obtained from batteries, fuel cells, solar panels or rectified DC from isolation transformers.

The converter requires no components with a voltage rating higher than that of a single cell. The converter is ideally suited for series voltage injection application and bypasses the need for a series injection transformer [27].

## 2.2 Basic building blocks

From the preceding discussion of the converter topologies, it can be seen that most of the converters can be constructed from a number of basic building blocks. To qualify as a building block, the circuit must be repeated within the converter and be controllable as a self-contained unit. All component values must preferably remain unchanged between placements within the converter. The following configurations are defined as building blocks: half-bridge, full-bridge and imbricated-cell. The blocks or cells are shown in Figure 2-16, Figure 2-17 and Figure 2-18.

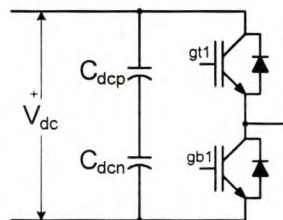


Figure 2-16 – Basic converter blocks (half-bridge)

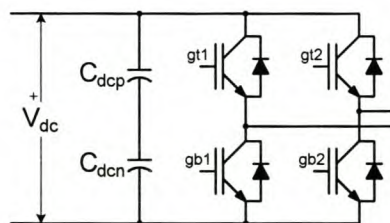
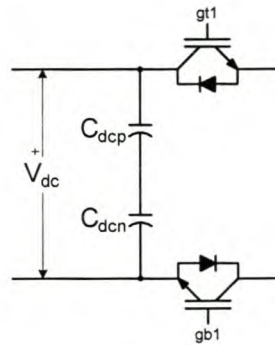


Figure 2-17 – Basic converter blocks (full-bridge)





**Figure 2-18 – Basic converter blocks (imbricated-cell)**

For the dissertation only one of the cells is evaluated. The choice of cell must be flexible in usage and application.

The half-bridge can be identified in a number of the basic topologies, but for the multilevel topologies, only the neutral point clamped (diode clamped) topology contains some form of the half-bridge. The half-bridge cell of the diode clamped converter can also not be controlled as a single unit, but rather the state all of the switches of a single leg of the converter must be switched as a unit.

The full-bridge converter can be identified in the majority of the multilevel and multicell converters. Two converters requiring special attention are the series stacked and cascade multilevel converters. The half-bridge can be identified as the basic building blocks of the converters. The added advantage is that the blocks are the same, with the same DC-bus voltage rating and current capability. This does not necessarily imply that this must always be the case, because some form of hybrid converter can optimize the voltage rating of successive blocks to achieve a certain response on the output [31].

The imbricated-cell is a repeatable block identified as part of the construction of the imbricated-cell multilevel converter. The main problem with the circuit is that the voltage rating of each adjacent cell is different and increases linearly with the number of levels of the converter. It is therefore impossible to construct a generic cell. The voltage rating not only applies to the DC-bus capacitors of the cell, but the gate-drive isolation and power supplies must also be rated at the highest voltage rating.

From the above discussion it is apparent that the full-bridge block is the most versatile of the basic building blocks identified. The full-bridge is therefore chosen as the fundamental power stage of the generic power cell used for the project.

## 2.3 Summary

The chapter introduced a number of converter topologies, ranging from very basic topologies to complex converters with many switches.

After investigating some of the standard and emerging topologies, a number of fundamental cells can be identified. Large-scale converters can be built by combining a number of these basic cells in different configurations.

From the discussion, three basic cells can be identified, listed below:

- Half-bridge or phase-arm
- Full-bridge (H-bridge)
- Imbricated-cell (flying capacitor)

Of the three identified cells, the full-bridge is the most versatile. Multilevel converters, built with the half-bridge cells, generally do not require different voltage ratings or DC-bus capacitance for individual cells. The main disadvantage of the flying capacitor topology is the increased voltage rating required for each additional cell of the converter.

The full-bridge converter was chosen for the power stage of the basic building block. The design, control and application of the building block are discussed in the following chapters.



# Chapter 3

---

Analysis of the distributed  
control strategy

## Chapter 3 Analysis of the distributed control strategy

### 3.1 Introduction

Section 3.2 of this chapter gives a brief discussion of the traditional method of controlling a converter or complete system, as in the case of an industrial power-quality compensator. This forms an introduction to the control of more complex systems, multilevel or multi-converter, and leads to the investigation of alternative methods of control of these converters.

In section 3.3 the distributed control strategy is analysed and a number of configurations for the module controller are identified. Each configuration requires a different set of measurements and control/status signals. The investigation further determines the minimum requirement for the module controller to enable operation in all of the modes identified.

### 3.2 Traditional control methods

This section investigates the typical controller requirements for a small- and large-scale converter system. The Dynamic Power-Quality Compensator (DPQC) is a three-phase converter (optional 4<sup>th</sup> phase) connected in shunt with the load. The PEC31 [25] (TMS320C31-based power electronics controller) is a controller designed for laboratory prototyping of converter systems and applications.

#### 3.2.1 Dynamic Power-Quality Compensator (DPQC)

The DPQC is a shunt power-quality compensator, capable of active power filtering voltage regulation and DIP compensation. The device can be interfaced to a number of energy sources, e.g. lead-acid batteries, flywheel energy storage or flow-cell batteries.

The energy storage enables the device to compensate for dips and outages of the supply. The system is an industrial unit and serves as a base for the discussion of the requirements, control and construction (measurements, control, etc.) of an industrial system. The topology is shown in Figure 3-1.



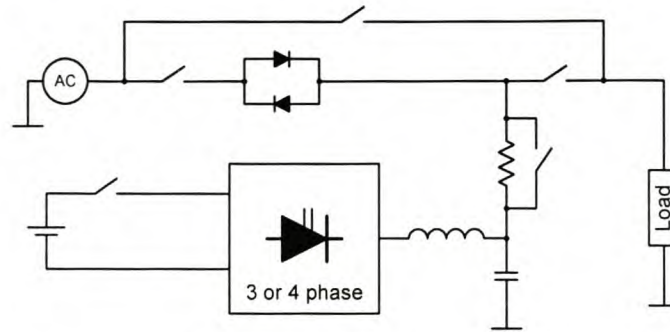


Figure 3-1 – DPQC topology

### 3.2.2 Series stacked converter utilizing PEC31 controller

The PEC31 (Power Electronics Controller C31) [25] is a custom controller designed primarily for the control of development and prototype systems. The initial design focused on the control of three- and four-phase converters, but the controller has been adapted for the control of multilevel converters like the cascade and flying capacitor multicell converters. The controller has also been used for the control of a three-level series stacked converter. These complex topologies have highlighted the shortcoming of a control/measurement system designed from the start to be flexible. Each additional topology needs additional control signals and measurements and gives rise to voltage isolation problems of the control signals and measurements. The need for an improved method of control exists and this leads to the investigation into distributed control. A case study of the series stacked converter is presented in section 3.7.1.

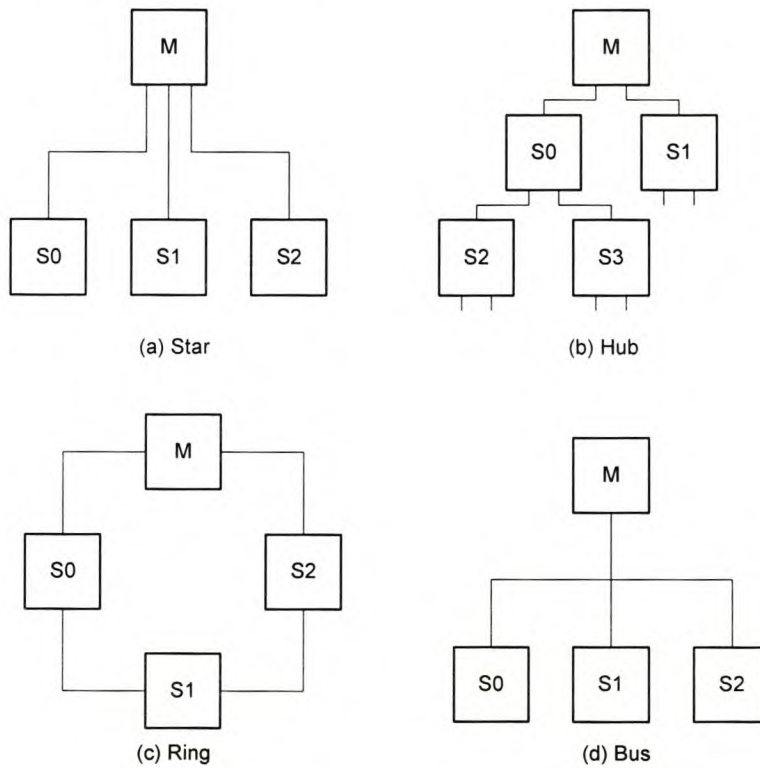
## 3.3 Distributed control strategy

### 3.3.1 Connecting the blocks

The distributed control strategy differs considerably from the traditional centralized control approach. Traditionally, all control signals originate from the central control board and all system variables are fed back to the controller. For small systems, this is still the preferred option. As the complexity of the system increases (for example, more switching levels or multiple converters), the amount of information from the converter increases dramatically and more control signals have to be generated.

The distributed control strategy splits the control between the various modules and/or converters. Instead of having a centralized controller that processes all the data, a

number of smaller controllers process data relevant only to the converter or module currently being controlled. For the system to operate as a single unit, the converters and modules are interconnected to enable the exchange of status and control information. For now no assumptions are made about the physical characteristics of the communications layer, as this will be discussed in detail in section 4.5. A number of interconnect configurations are possible and some of the more standard configurations are shown in Figure 3-2.



**Figure 3-2 – Module interconnect options**

The choice of interconnect configuration has a drastic effect on the number of communication links, the reliability (link and module redundancy), the required data bandwidth and the data latency. The four main configurations are described below.

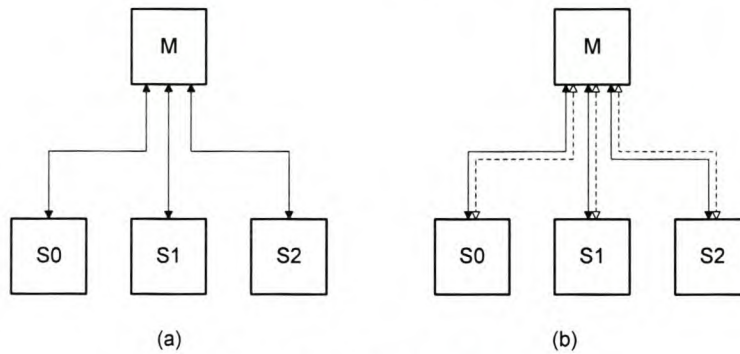
Depending on the physical medium used to communicate, a bi-directional link can be either a single bi-directional or dual uni-directional (simplex) link. This becomes an important issue for the final choice of layout, as this affects cost and reliability.

### Star connection

The star connection is probably the easiest configuration to understand and implement. Each module has a separate bi-directional data link dedicated to the



controller. The data bandwidth is therefore fixed, independent of the number of modules being used. The problem is that the main controller has to be supplied with a large number of link nodes and the system must be designed for a maximum number of modules. The requirements will vary vastly from system to system and a large number of redundant nodes will be wasted, leading to unnecessary extra cost.

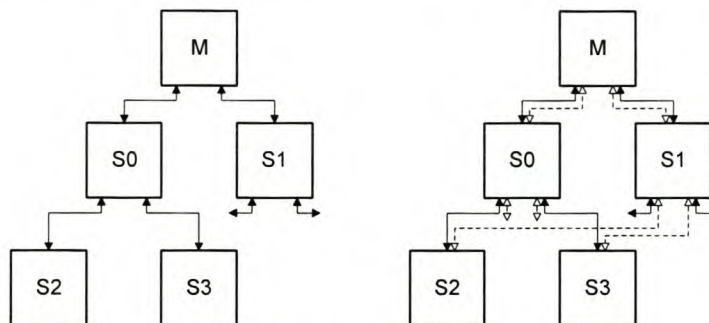


**Figure 3-3 – Star configuration**

The loss of a module or data link will disable the slave, but will not influence the operation of the other slaves. Redundant data links can be added by simply connecting a second link in parallel. Data latency is minimal and is not influenced by the number of modules in the system.

### Hub

Each module can be defined as a hub with one input and two output ports. The tree is then formed by connecting one or two modules to the preceding module or hub, as shown in Figure 3-4. The bandwidth requirement increases linearly with each additional module added to the lower branch. All links are bi-directional and the links of the bottom-most modules will not be utilized.



**Figure 3-4 – Hub configuration**

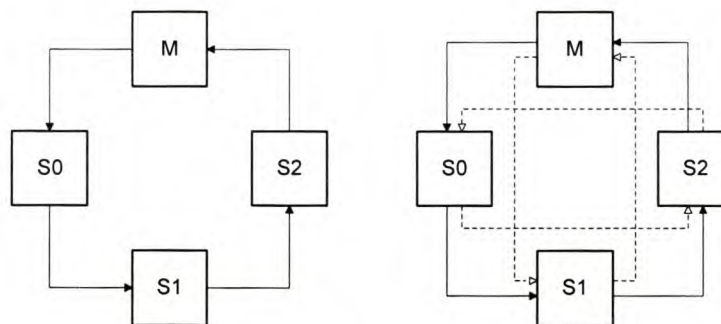
The loss of a module or data link will cause the lower branch to malfunction. Redundancy can be added by linking each pair to both modules higher up. The loss of

a single module or link can therefore be bypassed and, depending on the topology, operation can be continued.

The maximum number of modules is limited only by the data bandwidth and maximum allowable data latency from the main controller to the bottom-most modules.

### Ring

The communications ring is formed by connecting the output of each module to the data input of the next module (also called a daisy-chain). All the links in the chain are single direction only. The loop is closed by connecting the output from the last module to the input of the main module, as shown in Figure 3-5.



**Figure 3-5 – Ring configuration**

The loss of a data link or module will break the flow of data through the link. Data can be sent to all the modules upstream of the failed device or link, and data can be received from all the modules downstream from the failed device or link. Link redundancy can be added by using a secondary link that connects every second module together. If a module fails, an alternative path is available to feed the modules lower down the stream. The required bandwidth increases linearly with the number of modules in the link. The worst-case data latency is from the main controller to the last module in the chain. The maximum number of modules is limited by the channel bandwidth and maximum tolerable data latency.

### Bus

The bus configuration is only mentioned for completeness. All the devices connect to a common bus. The bus must be capable of bidirectional communication and voltage isolation is a problem. The bus configuration is therefore not suitable for this application.



### 3.4 Configuration options for module controller

This section shows how the module controller can be used in a number of configurations. The configurations are also analysed to determine a core set of requirements, as listed below:

- Type and number of communications channels
- Number of voltage measurements
- Number of current measurements
- Number of temperature measurements
- Number of gate-drive signals
- Maximum number of modules

The number of each of these subsystems will differ for different module configurations. The proposed controller must therefore be designed according to the maximum required number for each of the configurations.

#### 3.4.1 Current regulator configuration

The configuration for a current regulator is shown in Figure 3-6. The output of the phase-arm is measured and fed back to the controller. The DC-bus voltage is also measured and fed back to the controller. The second voltage measurement is optional and can be used for protection and/or control purposes.

The DC-bus voltage is used for protection and current control. The half-bridge can also be replaced with a full-bridge. This has the advantage of doubling the effective switching frequency, reducing the ripple and improving utilization of the DC-bus voltage.

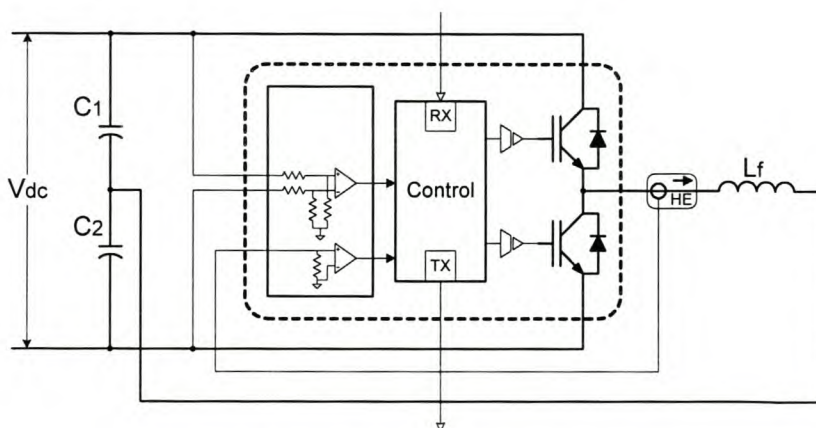


Figure 3-6 – Current regulator configuration

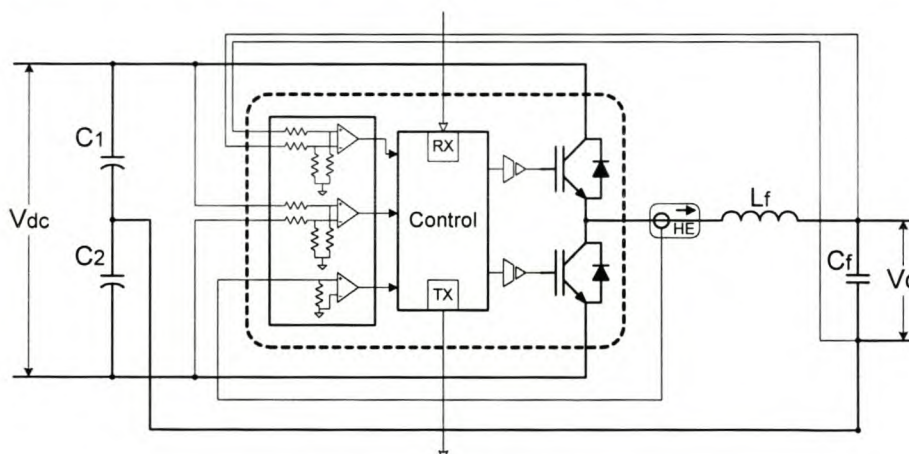
The requirements for the current regulator are summarised in Table 3-1.

**Table 3-1 – Current regulator requirements**

Subsystem	Number required
Voltage measurements	1
Current measurements	1
Temperature measurements	1
Gate-drives	2 or 4 depending on half/full-bridge

### 3.4.2 Voltage regulator configuration

The configuration for a voltage regulator is shown in Figure 3-7. The phase-arm output current is measured and fed to the controller. The signal can be used for protection as well as for a high-speed inner-current loop. The output voltage is measured for closed-loop control. The DC-bus is also measured for protection and possible control purposes. As in the current regulator configuration, the half-bridge converter can be replaced with a full-bridge converter for increased performance.



**Figure 3-7 – Voltage regulator configuration**

The requirements for the voltage regulator are summarised in Table 3-2.

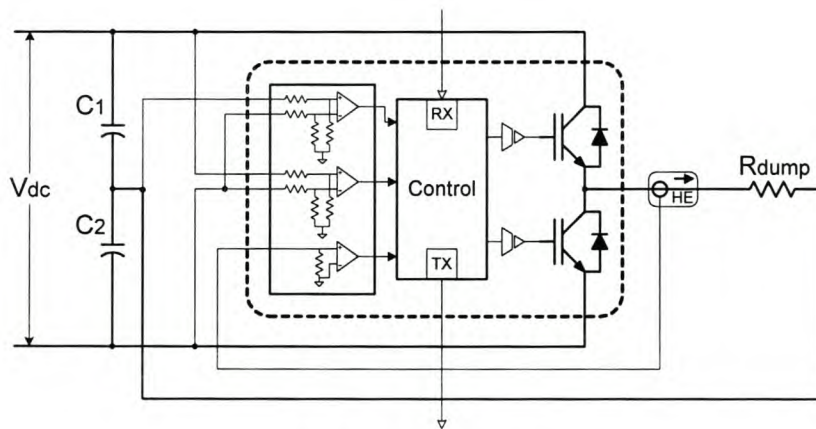
**Table 3-2 – Voltage regulator requirements**

Subsystem	Number required
Voltage measurements	2
Current measurements	1
Temperature measurements	1
Gate-drives	2 or 4 depending on half/full-bridge



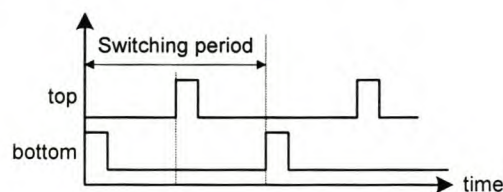
### 3.4.3 DC-bus over-voltage protection

The power-cell must be able to protect systems from DC-bus over-voltage conditions when connected to sources such as regenerative loads or the supply grid. The device is typically called a DC-dump and, as the name implies, energy from the DC-bus is dissipated in a resistive load. Traditionally, only a single switch is used to connect a resistive load across the DC-bus. Some of the topologies described in the dissertation connect conductors to the center tap of the split DC-bus capacitors. This can cause the DC-bus capacitors to become unbalanced. Different leakage currents or malfunctioning capacitors can also cause the voltages across the capacitors to become unbalanced.



**Figure 3-8 – DC-dump configuration**

A special configuration of the module controller can alleviate this problem by alternatively dissipating energy from the top and bottom capacitors into the resistive load. The topology is shown in Figure 3-8. The switches are fired the same as for a push-pull converter [1], rather than the standard phase-arm, as shown in Figure 3-9.



**Figure 3-9 – Gating signals for DC-bus over-voltage configuration**

The output current can be measured for protection and both capacitor voltages are measured for balancing and control. The requirements for the DC-dump configuration are shown in Table 3-3.

Table 3-3 – DC-bus over-voltage protection requirements

Subsystem	Number required
Voltage measurements	1 or 2
Current measurements	1 (optional)
Temperature measurements	1
Gate-drives	1 or 2

3.4.4 Capacitor center tap regulator

For three-phase four-wire converters, the star-point is connected to the capacitor center tap. Any zero sequence current will cause a neutral current to flow into the capacitor center-tap. The neutral current will cause the capacitor voltages to become unbalanced. A larger DC-bus capacitance will allow more unbalance to be handled, while remaining within the voltage rating of the DC-bus capacitors.

The three-phase four-wire topology was shown in Figure 2-5, section 2.1.2. By adding an additional phase-arm, the capacitor center tap voltage can be controlled, as was shown in Figure 2-7. The voltage of the individual capacitors are measured and controlled by regulating the output-current of the fourth phase-arm.

The configuration for the module controller for this topology is shown in Figure 3-10. The configuration is a combination of the voltage regulator and DC-dump configurations. This circuit cannot protect the converter from a DC-bus over-voltage, because there is no method of getting rid of excessive energy. Both capacitor voltages are measured, as well as the phase-arm output current.

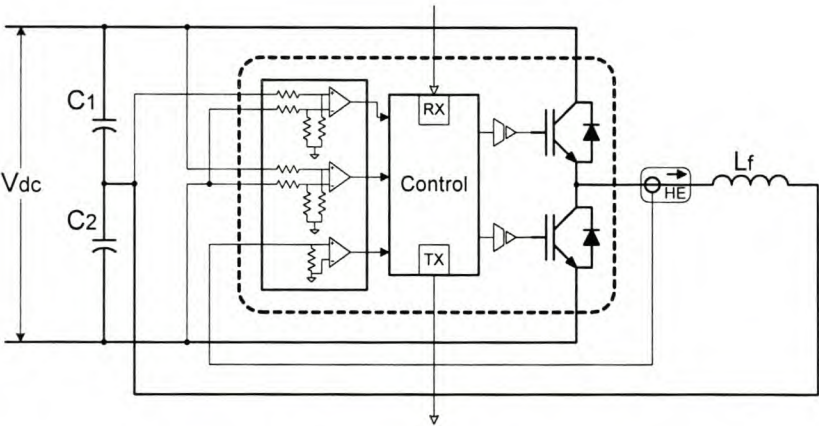


Figure 3-10 – Capacitor center tap regulator



The requirements for the capacitor center tap regulator are shown in Table 3-4.

**Table 3-4 – Capacitor center tap regulator requirements**

Subsystem	Number required
Voltage measurements	2
Current measurements	1
Temperature measurements	1
Gate-drives	2

**3.5 Proposed module controller**

This section describes the functionality of the proposed module controller and gives an overview of how the module controller can be used in the construction of a converter. The core set of subsystems is also determined by examining the requirements for each configuration described in the previous section.

**3.5.1 Requirements**

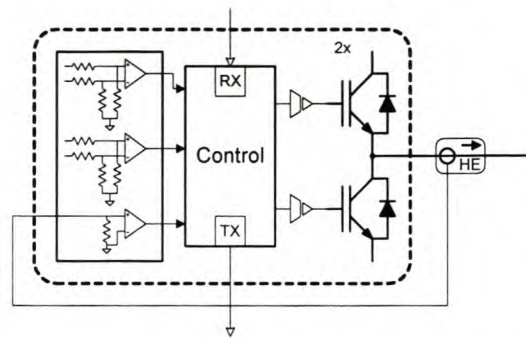
The core functions are the minimum set of requirements of the module controller to perform the various functions detailed in the previous section. From the tabulated values given for each configuration, the following set of functions was identified.

**Table 3-5 – Required functions of proposed controller**

Subsystem	Number required
Voltage measurements	2
Current measurements	1
Temperature measurements	1
Gate-drives	4

**3.5.2 Functional block diagram of proposed controller**

The functional block diagram of the proposed module controller is shown in Figure 3-11. A more detailed block diagram is given in section 5.2. The controller is capable of driving two phase-arms, enabling the full-bridge operation.



**Figure 3-11 – Functional block diagram of proposed module controller**

### 3.6 Main outer-loop controller

Not all systems need a complex outer-loop controller. A typical example would be a low-power uninterruptible power supply (UPS). For these cases the module controllers can be connected in a closed loop and set to run autonomously. Any errors that occur would be propagated through the communications loop and the modcons will shut down automatically. Depending on the severity of the error, the system can be set to automatically restart when the error clears. As large-scale UPSs require complex switchgear and more extensive energy storage control, a master controller would therefore typically handle the overall control of the system. An example of this is the DPQC mentioned previously.

The eventual target application of the modcon is for larger converter systems, where the construction and control of the system as a whole become a lot more complex. These systems generally require complex outer-loop control and an additional controller is necessary to act as the outer-loop controller, general system supervisor and possibly a human interface. This controller is designated the main controller and will always act as the master of the communications loop. A single master controller can also be used to connect with multiple modcon loops. This bypasses the inherent limit of the communications loop and allows the construction of more complex systems.

### 3.7 System configurations

This section shows how the module controller can be used in a number of test systems. The implementations are evaluated based on the number of interconnects, the controllability and the estimated reliability of the distributed control approach.



### 3.7.1 Shunt UPS using stacked converters

The stacked converter can be used to implement a UPS, where the DC supply is higher than would normally be required for a single converter. The higher DC-bus voltage has the advantage of lowering the current rating requirements. Typical applications like railroad traction use a high DC voltage in the order of 3.3 kV. Energy sources like super-conducting energy storage devices (SMES) also deliver a higher DC-bus voltage.

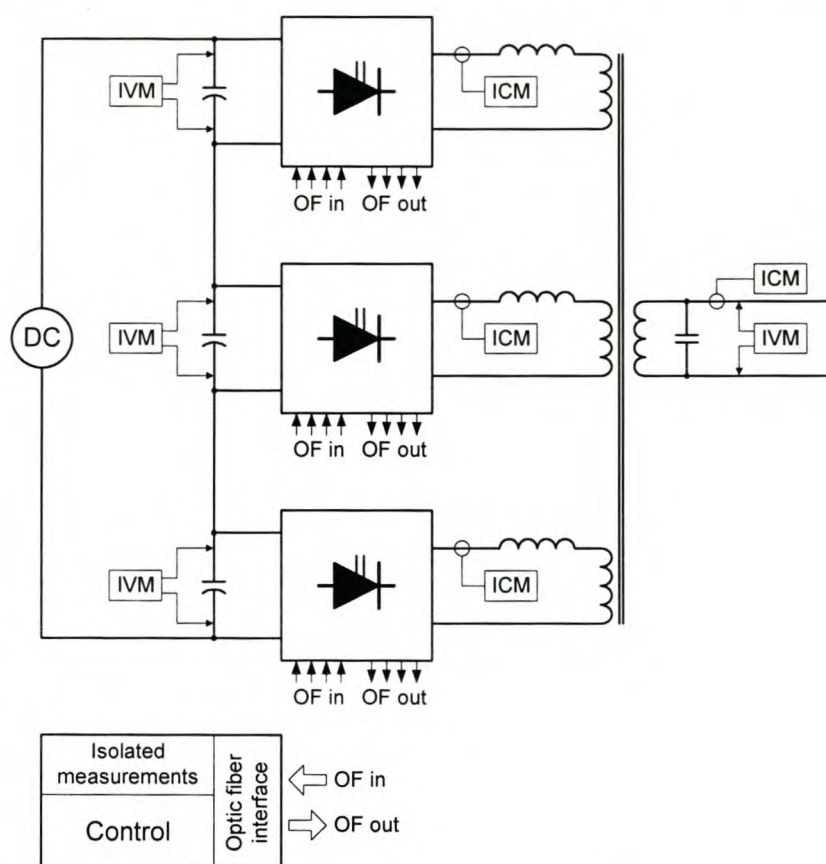
The example discussed here is for a shunt UPS, using three converters stacked in series. Only one of the phases is shown. A three-phase converter can be constructed using three of the converters shown in Figure 3-12 or, alternatively, the converters can be replaced with three-phase converters.

The discussion concerns only the converter and not all the support circuitry and switchgear needed for a complete industrial solution. As an example of an industrial application, refer to Figure 3-1 for the DPQC topology.

The single phase using the standard centralized control approach is shown in Figure 3-12. The number and type of measurements needed are determined by the application and the level of built-in self-test (BIT) required. Each cell has only an output filter inductor, with the output filter capacitor lumped together on the secondary side of the transformer.

The main controller must measure signals at different voltage potentials and therefore require isolated voltage (IVM) and current (ICM) measurement units. Depending on the switching action, the measurements will also have a high common mode content. It is therefore advisable to measure the voltage or current locally and send the result to the master via an optic fiber connection.

Each converter is a full-bridge converter, requiring four gating signals and returning one to four status signals. For industrial applications it is advisable to use a status signal per switching device to allow for improved testability.



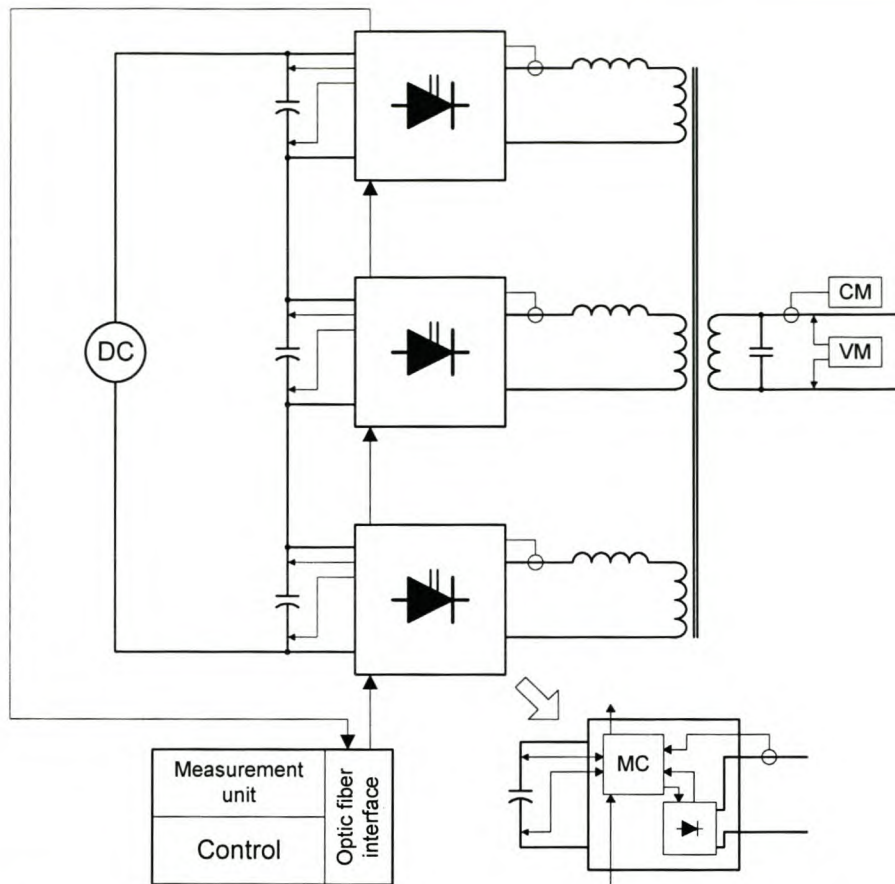
**Figure 3-12 – Single phase of shunt UPS using stacked converters (centralized control)**

The total number of connections is summarised in Table 3-6, where it is compared with the requirements for the distributed control approach. Advantages and disadvantages of using the centralized control are discussed in the following sections.

The construction of the same phase-arm using the distributed control approach (i.e. using intelligent power blocks) is shown in Figure 3-13.

Each cell is supplied with a local controller and measurement unit. The controller is only responsible for controlling the switches of the cell and measuring the local system variables (DC-bus voltage, output current, temperature, etc.). The combined output of the converter is measured by a separate measurement system. Depending on the ground reference, the measurements can be either local or isolated measurements. For the distributed control approach, there is the option of including the output measurement unit in the fiber communications loop. The measurement unit would be similar to the module controller, except for the additional circuitry required for switching the IGBTs.





**Figure 3-13 – Single phase of shunt UPS using stacked converters (distributed control)**

The requirements for the centralized and distributed control options are listed in Table 3-6. The current and voltage measure numbers do not refer to the physical measurement unit (current transformer, hall-effect current or voltage transducers, etc.), but rather to the measurement unit connected to the controller. The same number of physical measurement devices must be used (because the same number of measurements are used), but the type of measurement unit will differ.

For example, if the DC-bus voltage of the half-bridge is measured by the centralized controller, the measurement must be an isolated DC measurement. The common mode voltage of the measurement depends on the number of cells used.

If the same measurement is taken by the module controller, only a local measurement is needed, because the voltage reference is the same voltage used by the module controller. This signal will not have the common mode voltage problem and the required common-mode range does not depend on the number of cells used. The actual common-mode problem does not magically disappear, but the responsibility is shifted to the power supply of the module.



From Table 3-6 it can be seen that the requirements for centralized control are much more elaborate. The number of fibers increase greatly compared to the number of fibers for the distributed approach. As mentioned before, not all the measurements are strictly needed for operation in all modes of operation. The redundancy can be used to detect failing or broken measurement and control/status connections.

**Table 3-6 – Summary of connections for series stacked converter**

	<b>Control fibers</b>	<b>Status fibers</b>	<b>Voltage measure</b>	<b>Current measure</b>
Series stacked (Centralized)	12	12	3 IVM 1 VM (IVM)	3 ICM 1 CM (ICM)
Series stacked (distributed) with separate output measurements	4		1 VM (IVM)	1 CM (ICM)
Series stacked (distributed) with inclusive output measurements	5			

### 3.7.2 Series voltage regulator using a cascade multilevel converter

The cascade multilevel converter is an ideal solution for transformerless dip compensation [27]. The device operates by adding a voltage in series to the supply. The disadvantage is that an additional source is needed for charging the energy storage of the cells. This is because drawing active power from the supply will cause a dip in the load voltage.

The converter is constructed by cascading the outputs of a number of full-bridge converters. For a three-phase system, three separate single-phase converters are used. The converter using centralized control is shown Figure 3-14. Similar to the case study of the series stacked UPS, this discussion only pertains to the construction and operation of the converter and not to the support circuitry and switchgear required for the fully operational product.

The device operates by injecting a voltage in series with the supply. The main controller must therefore measure the supply voltage to detect when a dip occurs. The load voltage is not strictly needed, as this can be determined from the injected and supply voltage. The measurement allows for self-testing and calibration, and can be used to increase the performance of the device. The main advantage of using the cascade converter (especially if a high number of cells are used) is that the individual



cells can be switched at the fundamental frequency of the supply and need not be pulse-width modulated at a much higher frequency.

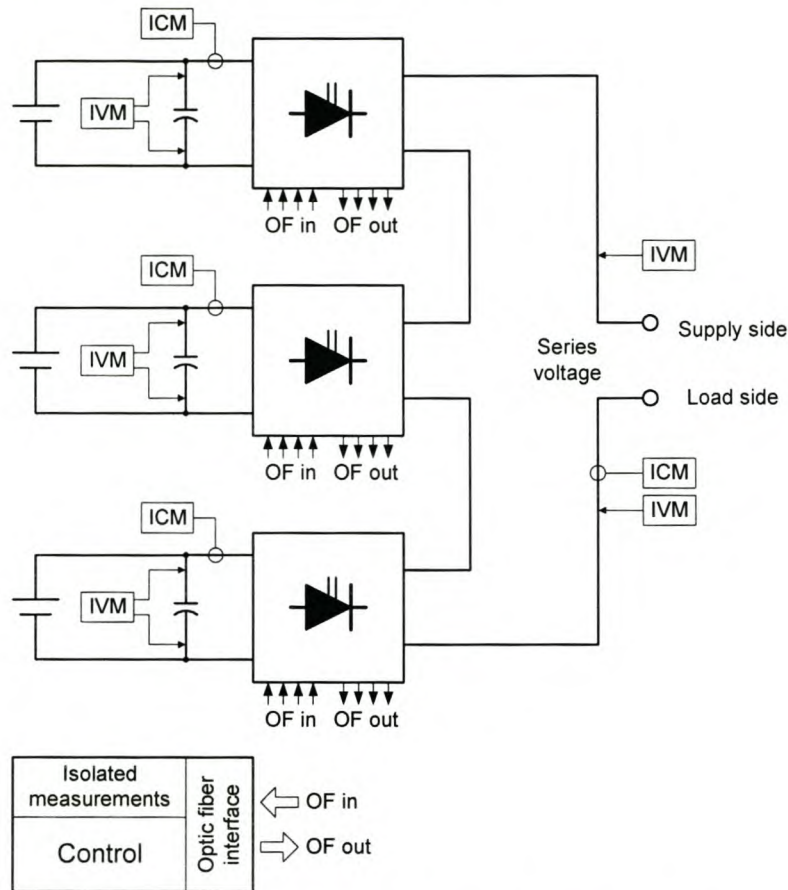


Figure 3-14 – Series dip compensator using cascaded converters (centralized control)

With the controller referenced to ground potential, the measurements must be isolated. The measurements will have a high common mode potential, which will differ according to the number of levels used for the construction of the converter.

The injection voltage is the sum of the individual cell voltages, although the current flowing through the device will be the same (the load current). It is therefore not possible to control each cell by using only the local measurements of the cell.

Each cell requires a separate DC-link and energy source. The main controller is responsible for sharing the energy usage from each cell [30]. This is especially important if the converter is operated at a low modulation index [29]. For low modulation indexes, there are a large number of redundant states available for balancing the utilization of the energy sources.

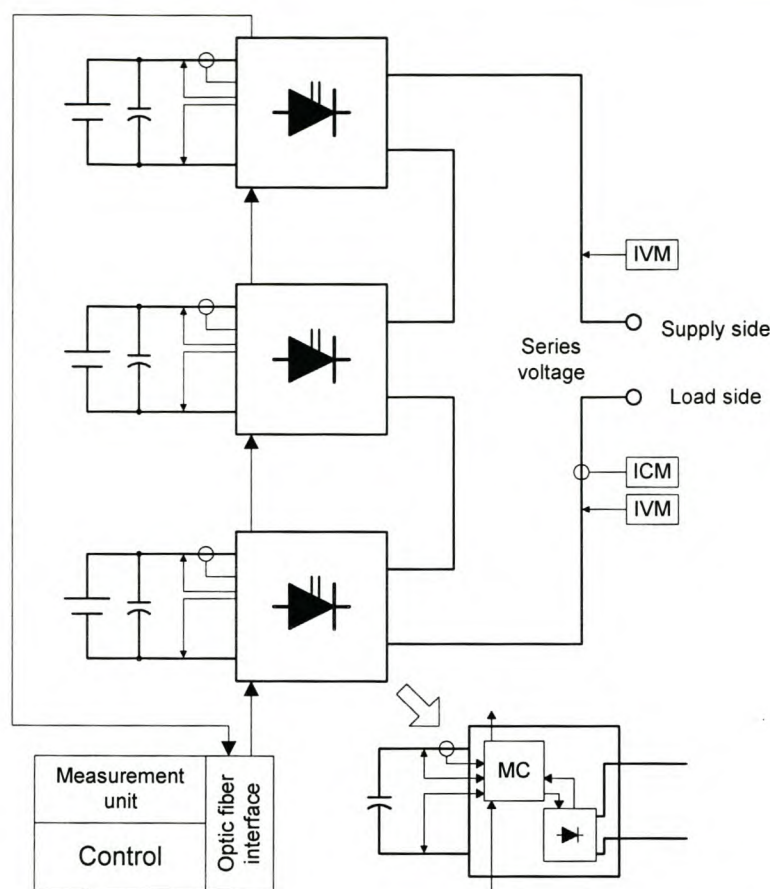


Figure 3-15 – Series dip compensator using cascaded converters (distributed control)

The summary (shown in Table 3-7) of connection requirements is very similar to the series stacked converter.

Table 3-7 – Summary of connections for cascade converter

	Control fibers	Status fibers	Voltage measure	Current measure
Cascade converter (Centralized)	12	12	3 IVM 2 VM (IVM)	3 ICM 1 CM (ICM)
Cascade converter (distributed) with separate output measurements	4		1 IVM 1 VM (IVM)	1 CM (ICM)
Cascade converter (distributed) with inclusive output measurements	5			



## 3.8 Analysis

### 3.8.1 Manufacturability

From a manufacturing view, the use of standard power blocks would ease the construction tremendously. The designer picks the required power/voltage/current rating, switching frequency and cell physical volume and constructs the converter from the individual power blocks. The manufacturer of the power block is responsible for guaranteeing that the power block will operate as specified. This is similar to ordering standard analog and digital integrated circuits from a catalogue.

### 3.8.2 Reliability

Large-scale systems using multiple converters or cells have a large number of control and status signals to drive and monitor the individual switches. It is not uncommon to use low-cost plastic optic fiber for the signalling, especially if high-voltage and large common mode signals are switched. Although the discussion will from now on refer only to the plastic optical fiber interconnects, the discussion also applies to other forms of interconnects, e.g. ribbon and twisted pair.

The electrical control signal is converted to light using a LED. The required power is calculated for the length of the fiber and the expected transmission baud rate. The receiver is a light-sensitive diode/transistor and converts the light intensity to an electrical value. The received signal is a small AC signal superimposed on a variable DC value. A quantizer is therefore needed to convert the signal to binary form.

A number of the low-cost optic fiber receivers include both the sensor/amplifier and quantizer circuit in the packaging. An example of this is the Agilent HFBR series of low-cost plastic fiber transmitters and receivers. The baud rate is usually limited to below 10 MBd. Higher baud rate links require more sensitive receivers and are usually AC coupled for increased sensitivity.

During the lifetime of the fiber, the plastic can degrade and become more opaque. This allows less light to pass through and can adversely affect the signal integrity. The problem is made even worse by badly polished fiber tips and tight corners when routing the fibers in the target system.

Control and status signals are usually used point-point, where the gate-drive is either on if there is light. The same holds true for the power device switch signal, although



it is preferable to use an active high signal for status OK. This allows the controller to detect a fault if the power switch power-supply fails or the fiber is disconnected.

After prolonged operation some of the control signals might lose integrity because of the above-mentioned problems. This can be difficult to detect from the controller, as operation of the converter can usually continue. The problem escalates as the converter becomes more complex (more levels/cells, etc.). Additional circuitry can be added to the switch drive circuit to detect excessive idle time or non-conforming control signals. Alternatively, the control and status signals can be modulated (e.g. frequency or pulse width) to check the signal integrity.

The use of the module controller with both the isolated power supply and communications link allows the control circuit to be situated in close proximity to the power switch. Additional data (e.g. status from slave devices) are sent via optic fiber to the master controller. The data is encoded and a Frame Check Sequence (FCS) is added to verify the integrity of data. This allows for a more reliable flow of information.

### 3.8.3 Controllability

The use of multiple, separate controllers allows for more detailed control of individual units of the large-scale system. This can be both an advantage and a serious cause for concern. Care must be taken not to disturb the inner dynamics of the converter. The interaction of cells within the converter has to be analyzed and possible unbalance situations handled by the controller.

The additional level of control provided by the module controller can sometimes not be used to the fullest extent. The cascade converter is a typical example of this scenario. The traditional mode of operation of the cascade converter is to switch each cell at the fundamental frequency of the output current or voltage. The switching times for each cell are determined by looking at the overall parameters of the system. If each module had to calculate its own reference signal, it would have to gather data from each of the cells in the loop. The master controller would therefore gather the data from the slave devices and write new reference values to the slaves.

Each cell of the cascade converter is supplied from a separate source. The source can be a rectifier, battery or any source of active power. If only reactive power compensation is needed, the storage can be replaced with only capacitors. If the



source requires additional control for charging or maximum power point tracking (e.g. lead-acid batteries and photovoltaic panels) the module controller can be used to perform this function.

The module controller (configured as a half-bridge or even a three-phase converter) is an ideal solution for the series stacked topology. The topology consists of multiple identical cells that can be controlled individually. The only global data needed is the total DC-bus voltage or DC-bus reference.

When the converter is controlled from a centralized controller, the assumption can be made that the DC-bus voltages will balance naturally due to the losses in the system [43]. This assumes that the same reference is given to each cell. If interleaved switching is used, the references must be updated accordingly (i.e. do not use the calculated reference for all the cells). If the same reference is used, it will introduce a phase shift between the references, leading to an unbalanced voltage.

The distributed control version of the series stacked topology allows for the low-level control of each individual cell. If the converter is used as a current regulator, the unbalanced currents forced by the individual cells can dominate the natural balancing of the converter. This can easily be caused by offsets in the current measurements or discrepancies in DC-bus capacitor and filter component values.

Each cell can only directly measure the local voltage of the DC-bus, the injected current and one additional voltage. It can be argued that the second voltage measurement can be used to measure the total DC-bus voltage, but this would not always be feasible because any number of cells can be stacked in series and the on-board voltage measurement has a limited common mode and differential mode measurement range. The cell therefore requires data external to the cell, be it the total DC-bus voltage (determined by adding the individual measurements) or a DC-bus reference voltage.

### **3.9 Summary**

This section introduced the concept of distributed control as applied to power electronic converters. Examples were given of systems using the classic centralized approach and were compared with the distributed control approach.

Options were given for connecting the modules together in a communications link.

The three viable options are:

- Star
- Hub
- Ring

Each configuration has advantages and disadvantages. The criteria are:

- Number of links
- Bandwidth requirements
- Channel latency
- Reliability/redundancy

A module controller was proposed that is capable of operating in a number of modes or configurations. This allows for a very flexible power block that can be used in a variety of topologies and applications. The modes of operation are:

- Current regulator
- Voltage regulator
- DC-bus over-voltage protection
- Capacitor center-tap regulator

Two example systems were shown using distributed control. The discussion evaluated the number of control signals required, reliability, signal integrity detection and voltage isolation problems.



# Chapter 4

---

Interface requirements of the  
module controller

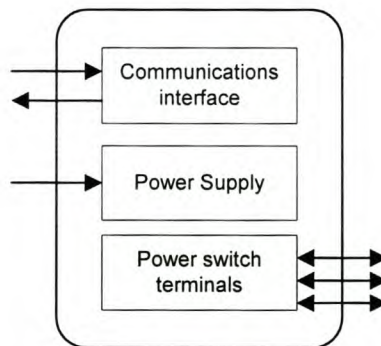
## Chapter 4 Interface requirements of the module controller

### 4.1 Introduction

This chapter discusses the choices and trade-offs that must be made when establishing the interface standards for the module controller. The discussion does not deal with technology-specific implementations, but rather deals with a number of basic concepts for interfacing to the module controller/power device. The detailed design of the module controller is discussed in Chapter 5.

### 4.2 Interfaces to the module controller

Figure 4-1 shows the interfaces required for the module controller to be used in a distributed control environment.



**Figure 4-1 – Module controller interfaces**

The three main interfaces are:

- Power switch terminals
- Power supply
- Communications link

The different interfaces will be discussed in more detail in the following sections.

### 4.3 Power switch terminals

The power terminals include the connections for the switch outputs and the DC-link terminals. The power switch terminals and physical housing of the module controller and power switch determines the physical layout and construction options of the converter or system as a whole. The layout can be similar to the standard modules available from the mainstream power module manufacturers like Powerex and Semikron. An alternative and possibly more attractive solution is the hockey puck



configuration. This will allow devices to be easily stacked for high power and voltage solutions.

#### **4.4 Power Supply**

The power supply is probably the sub-component of any system that is most easily overlooked. The power supply for the module controller is even more important, because the design can easily limit the end user in the choice of converter topology that can be realized with the power block.

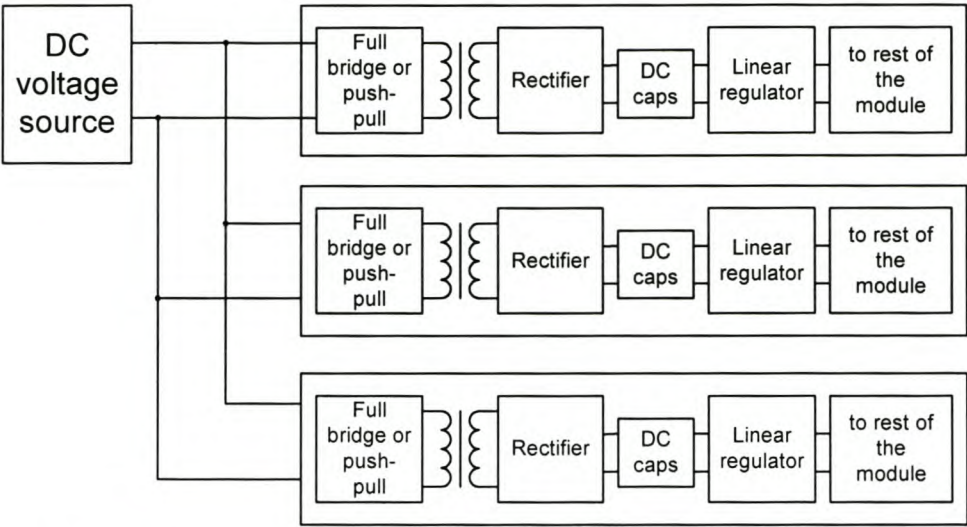
The power supply is not only restrictive when using distributed controllers, but even with traditional converter implementations. It is common practice to use local gate-drive circuitry situated close to each power switch. Voltage isolation of both the power supply for the gate-drive and the control and status signals is a problem. Conventional two-level converters are easier to interface, because the gate-drive circuits will be at known voltage levels relative to the controller ground. As mentioned previously, the developer must not be significantly limited in the choice of converter topology, even when stacking or chaining the blocks to form multilevel converters. The power supply should therefore have a sufficient voltage isolation capability to handle a predetermined number of levels.

There are a number of implementations available for the power supplies, not only the individual components, but also the means of transporting the power to the module. One limiting factor that will be present in most of the target systems is the requirement that the module controllers must be powered before the rest of the system can go online. This limits the actual source of power for the supply (batteries, mains, etc.).

A common implementation is to have a central power supply that distributes power to the individual gate-drive circuits. The power can be transmitted as a DC or an AC source. In order to keep the cost low, the DC implementation is usually a regulated supply with a fixed 50% duty cycle switched-mode power supply at each gate-drive. A linear regulator is then used to for the final regulation of the power supply. This method minimizes the filter components required at each gate-drive circuit and requires a very simple circuit. An alternative would be to put a regulated switch mode power supply at each gate interface, reducing the regulation limits for the distributed

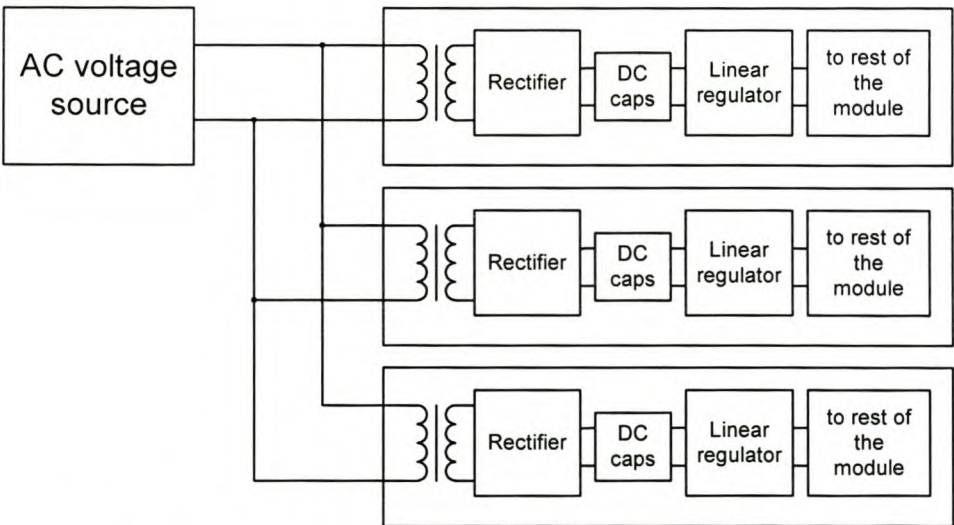


DC source, but requiring a more complex circuit at each gate-drive (for example, isolated feedback and inductive filter components).



**Figure 4-2 – Power supplies for controller and gate-drives (DC distribution)**

With the AC implementation the main power supply is a separate switch mode supply capable of handling the power requirements of all the gate-drive circuits and controllers. The main advantage is that the power supply circuitry at each gate-drive is very simple, consisting only of an isolation transformer, rectifier, DC reservoir capacitor and usually a linear regulator. A diagram of the circuit of the power supply is shown in Figure 4-3.



**Figure 4-3 – Power supplies for controller and gate-drives (AC distribution)**

A number of choices must be made with the AC distributed power supply. The first is the switching frequency and shape (square, sinusoid, etc.) of the AC waveform. This



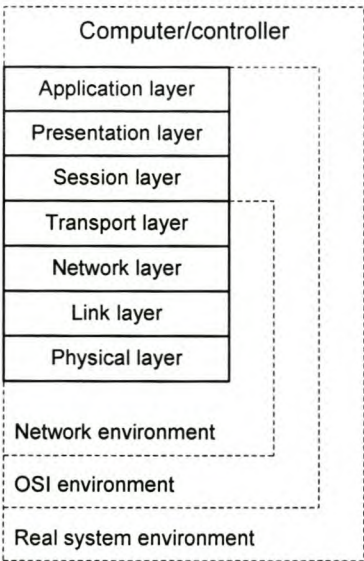
will have a major effect on the radiated EMI, as well as the size and design of the isolation transformers and filter components used for the local power supplies.

The ideal situation would be to have power supplies that are powered directly from the output or input of the power block. These implementations would alleviate the requirement for high-voltage isolation and would place virtually no limit on the choice of topology or configuration. The obvious problems with this configuration are the bootstrapping of the supply and the wide input range requirement of the supply.

**4.5 Communications interface**

The communications interface forms a very important part of the module controller and together with the power supply and power-terminals establishes a standard for interfacing to the power block.

This section investigates a number of standard communications interfaces and defines a custom communications interface for the module controller by building on the ISO reference model for communications subsystems [3]. The layered reference model is shown in Figure 4-4.



**Figure 4-4 – ISO reference model for communications subsystems**

The full reference model consists of seven layers, of which the first three layers are network specific. The fourth layer (transport layer) forms the interface between the lower network environment and the higher application layers.

#### 4.5.1 Serial communications background

This section discusses some of the important aspects and terms used when describing various serial communications standards.

##### Synchronous, asynchronous, isochronous and plesiochronous:

*Asynchronous* is generally used to describing events that are not coordinated in time. For communications systems an asynchronous signal is one that is transmitted at a different clock rate than another signal.

*Synchronous* describes events that are coordinated in time. When synchronous transmission is used for transmitting data, this means that all the clocks are referenced to a single clock. The clock and therefore data are thus in synchronization.

*Isochronous* means equal in time and pertains to processes that require timing coordination to be successful, such as voice and digital video transmission. An example is voice data going from a source to a destination across a transmission channel. The same rate of data flow is needed at the source and the destination.

*Plesiochronous* is used to describe operations that are almost, but not quite, in synchronization. Such a case could arise as the result of two systems having slightly different clock frequencies. To correct for this, one of the systems must make a slight adjustment. For data transmission this means that a data unit might have to be inserted or deleted.

##### Synchronous versus asynchronous transmission:

According to [3], the strict definition of asynchronous transmission is when the transmitter and receiver clocks are running asynchronously to each other while the data is being transmitted. Generally, it also indicates that data is being transmitted as individual characters or bytes, including a start bit and possibly a stop bit. The start and stop bits are used for synchronization.

##### Physical layer

The physical layer consists of the physical medium as well as the electrical interface.

The physical medium determines the EMC susceptibility, link length and voltage isolation capabilities of the communications link. The link for the module controller is a closed system and therefore RF transmission is not an option.



The standards mentioned in section 4.5.1 run on one of three physical mediums. These are copper conductor, optic fiber and through-the-air infrared. Reliability excludes the infrared transmission, therefore only copper conductor and optic fiber will be considered. The optic fiber link provides high-voltage isolation and the physical medium is not susceptible to EMI. The copper conductor-based link can be used with isolation transformers to provide high-voltage isolation. The physical medium is more susceptible to electromagnetic interference and the transformer solution requires an AC-coupled signal and therefore continuous transmission. Depending on the topology, the common mode voltage on the transformer can be very high and change with very high  $dV/dt$ . The switching of the converter is stochastic and therefore the high  $dV/dt$  common mode changes cannot be predicted and compensated for. This will adversely affect the data integrity; therefore the optic-fiber link was chosen for the final design.

There are two main classes of optic fiber, namely glass and plastic. These are further categorized as single or multi-mode. The module controller requires a low-cost link with relatively short lengths of fiber. The standard plastic optic fiber was therefore chosen for the link.

### Link layer

The link layer builds on the physical layer and is responsible for the reliable transmission of data across the physical medium. The connection can be either connectionless or connection oriented. The connectionless version uses a best-try approach and if errors are detected in the received data, the data is simply discarded. The connection-oriented approach tries to provide error-free communication by adding retransmit functionality. The limited bandwidth and strict timing requirements of the module controller communications link leans more towards a connectionless implementation; therefore erroneous data is simply discarded within tolerable levels.

### Data transmission

All module controllers are assumed to be running asynchronously, with a local clock generated on board for control and timing purposes. The assumption is made that a clock signal of 40 MHz (1% tolerance) is available for the communications link and can be independent of the controller clock.



There are two types of communication links, defined as either asynchronous or synchronous. A typical example of asynchronous transmission is the RS232 serial link found on most personal computers. This type of transfer is usually used when data transmission is intermittent, for example, the keystrokes of a person typing on a keyboard. The clocks of the transmitter and receiver run asynchronously. When a byte of data is sent, the receiver synchronizes with the incoming data by detecting an additional start bit that precedes the transmitted data, called bit synchronization. The rest of the data is then merely clocked in at specific timing intervals. The local clock typically runs at 16 times the transmitted data, so slight variations in the clock frequencies can be tolerated. The length of the data word has a significant influence on the maximum tolerable frequency error. For 16 times over-sampling, a maximum data length of 8 bits is advised.

For synchronous transmission it is assumed that the clocks of the transmitter and receiver run synchronously. The distinction between synchronous and asynchronous transmission is sometimes not clearly defined. An alternative distinction is that there is no start bit transmitted for synchronous transmission, but that data is transmitted as contiguous bits.

Synchronization of the clocks can be done in a number of ways, the simplest of which is to send the clock signal on an additional line. For the module controller the extra signal will require an additional optic fiber and is therefore not a viable option. The second option is to encode the clock signal as part of the data (forcing signal transitions for each data bit transmitted) and then extracting the clock signal at the target system. An example of this is Manchester encoding [3].

A third method is to use a digital phase locked loop (DPLL) on the received data. The DPLL generates a local clock that is used to clock out the data from the incoming stream. The input clock of the DPLL typically runs at 32 times the transmitted data, which can be significant if the transmission rate is high. In order for the DPLL to maintain a lock on the data stream, sufficient signal transitions are needed. The data is therefore encoded to ensure sufficient transitions in the data stream, typically at least one transition for every 5 bits. The PLLs require a certain amount of time to lock onto the clock signal (acquisition time). It is therefore advantageous to have continuous transmission of data, even when there is no data to send.



There are also hybrid schemes that combine the two above-mentioned techniques, for example, Manchester encoding (at least one transition for every bit transmitted) and the DPLL. The input and local clock ratio can be dropped below 32. The above-mentioned methods differ in their utilization of channel bandwidth. The number of signal transitions per second is measured in baud. The bit rate can be higher than the baud rate if multiple bits are represented by a single entity, for example, the phase and the level of the signal can represent data, both of which can change simultaneously.

For the embedded clock transmission there is more than one way of extracting the clock and data. The low-cost method is to use a sampling clock of around 8 times the data rate. A programmable logic device (PLD) then extracts the edges from the sampled input data. The edges are then used to generate the bit-sampling clock. For relatively low data rates (less than 5 MBd), this is a viable solution. As the transmission rate increases to 20 MBd and above, the sampling frequency required becomes excessively high and requires the use of an alternative approach for clock and data extraction.

When the channel bandwidth is much higher than the required baud rate, alternative schemes can be used, for example, pulse width modulation (PWM). As mentioned previously, this type of scheme does not utilize the full data transmission rate capability of the channel, but the transmitter and receiver circuits are much simpler.

### Transport layer

The protocols can be divided into two main categories, namely message-based protocols and address-based protocols. This is only relevant for communications systems that are capable of addressing multiple devices. Two systems connected point to point with an RS232 serial link would not require the additional overhead of an address-based protocol, unless multiple client devices can share the channel.

In a message-based communications link, the master does not send data to a specific device; rather it broadcasts the data (message) to all devices and only the devices that understand the message will accept the data. This can be an advantage; for example, a shutdown command can be broadcast to all devices. On the downside, all slave devices must receive and interpret all the transmitted messages and additional information must be added when more than one device of the same type resides in the



communications loop. A practical example of this is UDP (User datagram protocol), where the data packet is broadcast on the LAN or WAN.

With the address-based scheme, the master transmits the data to a specific slave by adding the address of the target to the packet of data. The advantage of this is that data can be sent to a specific target (for example, a current reference update). The downside is that broadcast messages are not possible. During an emergency shutdown, a command must be sent to each target separately, but this might not be a problem if the delays are acceptable. A practical example of such a protocol is TCP (Transmission control protocol) used in LANs and the Internet. Each target is uniquely specified by a 32-bit address called an IP. The inclusion of an additional 16-bit port number allows for addressing specific sub-systems within the target (multiple connections, processes or applications). The address-based protocol also has advantages with larger-scale networks, where routers can determine the exact path of the data packet, freeing bandwidth on the other channels.

The transmission rate required for the distributed control approach will vary widely depending on the converter topology and the level of control required. Some applications can have a module controller based converter acting as a UPS and running virtually autonomously. The main controller is then only responsible for sending an RMS voltage reference and gathering status information.

At the extreme opposite of the implementation, the main controller can send the duty-cycle information to each sub-module in the UPS for each switching cycle, while gathering system information such as load and supply voltages. This approach requires a lot more bandwidth than the previous approach.

For speciality cases the two methods can be combined to form a hybrid-addressing scheme. The address can be split into multiple fields, e.g. group and device address fields. For a target to respond to a packet of data, both the group and device address fields of the incoming packet must match the appropriate fields of the target. Specific device addresses can be reserved for broadcast packets. This approach was used for the module controller and will be discussed in more detail in section 5.4.



## 4.5.2 Standard communications interfaces

A number of existing serial communications interfaces are already in use in commercial and industrial equipment. Some of the more widely used standards are listed in the following section.

The links run on a wide variety of physical mediums and protocols. The requirements for the module controller are very specific and only a subset of the functionality of the above-mentioned standards is needed. The distributed control approach has a few additional requirements that some of the standards do not fulfil.

### 4.5.2.1 MACRO (*Motion and Control Ring Optical*) [48]

The link uses a twisted pair or glass fiber to transmit data in a ring structure. The maximum data rate is 125 Mbit/s and the latency associated with a device is approximately 1  $\mu$ s. The total number of devices is 16 per master, with a maximum of 16 masters per ring. MACRO seems to be an ideal solution, except for the relatively high price of the glass fiber based optic fiber link (approximately \$65.00 according to [48]).

### 4.5.2.2 SERCOS (*Serial Real-time Communication System*) [49]

The link is used for motion control systems and uses a fiber optic link for data transmission. The main problem with the link is the relatively low speed of 1 Mbit/s. The plan is to introduce a faster version at the start of January 2002, with speeds of 2/4/8/16 Mbit/s. The link has the ability to perform cyclic transfers for real-time application (e.g. the current reference updates and PWM references that must occur periodically), but the minimum cyclic rate is currently 62.5  $\mu$ s.

### 4.5.2.3 CAN (*Controller Area Network*) [51]

CAN is a serial bus system with multi-master capabilities, running on a common bus. All CAN nodes are able to transmit data and several CAN nodes can request the bus simultaneously. The serial bus system with real-time capabilities covers the lowest two layers of the ISO/OSI reference model. The protocol is message based (with individual priorities). Currently, the maximum speed is 1 Mbit/s.



#### 4.5.2.4 USB (*Universal Serial Bus*) [52]

USB is a serial communications link connected in a hub configuration. The currently available version is USB1.1, with a maximum transfer speed of 12 Mbit/s. USB2.0 will allow for data rates up to 480 Mbit/s. The link allows for isochronous and asynchronous transfers. The link is intended as a low-cost solution for connecting peripheral equipment to personal computers. Currently, there is no optic fiber version of the link available.

#### 4.5.2.5 IEEE 1394 (*Fire wire*) [50]

Fire wire is a serial link with a tree network topology. The link can use both twisted pair and optic fiber. Transfer modes include isochronous (125  $\mu$ s) transfers for real-time applications and asynchronous transfers. A maximum of 63 devices can be connected. According to [49], the update rates 4 kHz for position and velocity loops and 16 kHz for current loops.

#### 4.5.2.6 IrDA (*Infrared transmission*) [53]

IrDA is an asynchronous communications link making use of infrared signals to communicate with slave devices. The transfer rate ranges from 9600 Bd to 4 MBd, with a 16 MBd also defined. The link is intended for connecting peripheral devices to mobile and personal computer equipment.

### 4.5.3 Synchronization and data transparency

For communications sub-systems where only the data is available and no additional signals are used for synchronization (e.g. frame sync, etc.), some method must be established to detect the start and end of a packet of data. Special characters can be reserved for flagging the start and end of a frame of data.

This section describes some of the methods to obtain data transparency when using an asynchronous communications link like RS232 or IrDA. A method is needed to determine the start and end of a frame of data. The above-mentioned links can only send byte length characters (5-8 bits per data unit for the standard UARTs used for RS232, e.g. 16550). The control flags must therefore be embedded in the data stream. There is more than one implementation, but they all fall under the same category known as byte or character stuffing. Two standard methods are described below.



#### 4.5.3.1 RS232 character stuffing

A number of control characters are defined for the RS232 protocol and some of the control characters relevant to this discussion are listed below:

- DLE Data link escape character
- STX Start of text character
- ETX End of text character

If the data to be transmitted is purely text, the control characters will never appear in the data stream. Sending an STX character signals the start of the frame, while an ETX character signals the end of the frame. If the data is pure binary, the STX and ETX characters can appear as part of the data stream. A method must therefore be devised to make the character stuffing transparent to the receiver.

The character DLE is used to signal that a control character will be sent; therefore the start of a frame is signalled by a DLE/STX pair and the end of the frame by a DLE/ETX pair. If a DLE character appears as part of the data stream, a DLE/DLE pair is sent. The receiver checks for a DLE character and, if a second DLE character follows, one of them is dropped.

This method has the drawback that the length of the frame is dependent on the content of the data transmitted. The worst case is if the data stream consists of all DLE characters. For sending large packets of data, the data can be compressed (for example, run length encoding) and this will alleviate the problem mentioned above.

To summarize, a DLE character always signals a special character to follow (except for another DLE). For this reason additional control character can be defined for other purposes, without complicating the state machine or influencing the size of the packet of data.

#### 4.5.3.2 IrDA character stuffing

IrDA character stuffing differs from RS232 stuffing in the choice of frame delimiters and interpretation of the characters. The relevant control characters are:

- BOF Beginning of frame
- EOF End of frame
- CE Control escape



With IrDA character stuffing, the control characters are not preceded by a special character. The start of a new frame of data is simply signalled by transmitting a BOF character. Multiple consecutive BOF characters can be sent to allow the receiver to synchronize. Sending an EOF character signals the end of the frame of data. Unlike the RS232 stuffing, a DLE character does not precede the control characters. A BOF character that arrives at the receiver always signals the start of a new frame of data.

The IrDA stuffing scheme achieves data transparency by modifying the content of the data to be transmitted. Similar to the RS232 scheme, the IrDA defines an escape character, the CE (control escape) character to achieve data transparency. To detect a control character, the receiver must check for each of three control characters. The RS232 data transparency scheme, on the other hand, only has to check for the DLE character. If a BOF, EOF or CE character is found in the data stream, the transmitter inserts a CE character and complements bit 5 of the character (XORs the byte with 0x20). This should only occur for the BOF, EOF and CE characters.

To summarize, BOF and EOF characters always signal the start and end of a frame of data. If a CE character is received, the character is discarded and the next character to arrive is stored with bit 5 flipped.

This scheme was used to achieve data transparency on the asynchronous serial link of the module controller.

#### 4.5.4 Requirements of the communications link

The requirements of the module controller communications link are:

- Low latency
- High integrity
- Low cost
- High-voltage isolation
- Low-power requirements
- EMC hardened
- Layered for compatibility and expansion.

*Latency and bandwidth:* Not all of the requirements are strictly needed for every application or implementation. Low latency is only required for applications with a high control bandwidth. Self-contained, low-power UPS type applications require only slow synchronization and reference updates. The typical power electronic



application has more than one clock domain or time scale. The power electronics switch at a high frequency (typically in the order of 5 kHz), while the reference is at a much lower frequency (usually a 50 or 60 Hz sinusoid). Higher dynamic applications like active power filters generate higher frequency references, but these are still synchronized to the system power frequency of 50 or 60 Hz.

The use of a central controller has the advantage that all the PWM carriers are perfectly synchronized. This allows for optimal ripple switching because the zero states of the converter can be fully utilized. A method must therefore be devised to allow for synchronizing the carrier frequencies of the target controllers. This is a problem if the only link available is a low-frequency RS232 based protocol. The synchronization process is discussed in more detail in section 5.4.

*Voltage isolation and EMC:* The module controllers operate in a harsh EMI environment. The communications link is routed between modules with a wide variety of ground potentials that change with a high slew-rate. The level of radiated RF energy is also high. The link must therefore be able to operate in such a harsh environment.

*Power requirements:* The power for the link must be drawn from the isolated supply of the module controller. The power required must not place too high a demand on the isolated supply, as this is usually very limited.

*Layered implementation:* The communications sub-system must be layered with different layers interacting on different platforms. This allows parts of the link to be replaced with a different unit (speed, isolation, etc.) without affecting the operation of the link. The host software can communicate with the target devices through more than one type of link simply by selecting a different driver for the physical link.

## 4.6 Summary

This chapter discussed the requirements for interfacing to the module controller. The main areas of concern are:

- Power terminals
- Power supply
- Communications link

The options for the power supply were given and fall into one of three categories:

- Distributed DC
- Distributed AC
- Local DC or AC tapped from the DC link or output voltage

A brief overview of a number of standard serial interfaces was given and options for the communications interface were discussed. The requirements of the communications link were given and discussed.

The design of the module controller is detailed in the next chapter.



# Chapter 5

---

## Design of the module controller

## Chapter 5 Design of the module controller

### 5.1 Introduction

This chapter builds on the discussion of Chapter 4 and describes the specific implementation of the module controller used for the experimental evaluation of the distributed control strategy. The evaluation follows in the next chapter.

The design of the module controller entails both the hardware and the software needed for control and debugging.

### 5.2 Detailed block diagram of module controller

Figure 4-1 shows the detailed block diagram of the module controller. The controller consists of the communications link, power supplies, measurement system and gate-drive isolation.

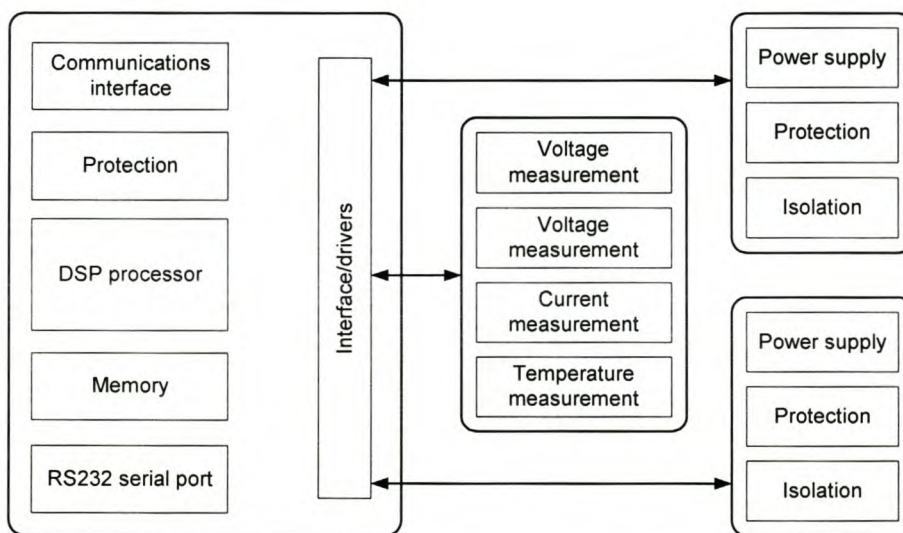


Figure 5-1 – Detailed block diagram of module controller

The specific implementation was designed to be modular and allow for easy testing and evaluation of the individual subsystems of the power block. The implementation consists of a number of boards, listed below and shown in Figure 5-2.

- Unregulated power supplies
- Gate interface and supply regulators
- Controller main board
- Controller measurement board
- Power supply main bridge



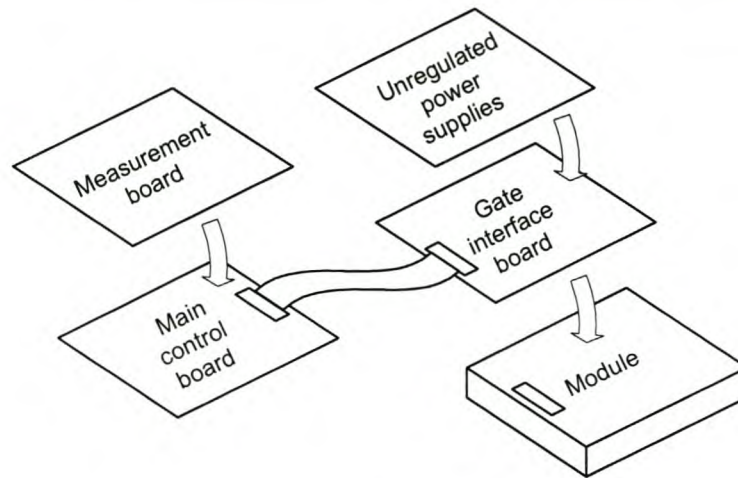


Figure 5-2 – Interconnection hierarchy

### 5.3 Controller and gate-drive power supplies

The options for the power supply were discussed in section 4.4. The distributed AC source was chosen for the specific module controller implementation, as this requires only one main bridge for the power supply.

A number of choices must be made with the design of the distributed AC power supply. The first is the switching frequency and shape (square, sinusoid, etc.) of the AC waveform. This will have a major effect on the radiated EMI, as well as the size and design of the isolation transformers and filter components used for the local power supplies.

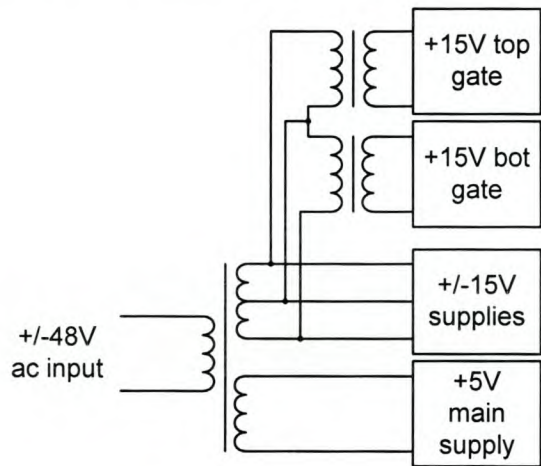
The choice was made to use a 250 kHz square wave. This leads to minimal passive filter requirements and the switching frequency is a trade-off between power losses in the isolation transformers (3F3 material) and the switches used for the switching supply (MOSFETs).

To prevent saturation of the isolation transformers due to voltage unbalance, the switching converter must deliver a balanced AC waveform to the isolation transformers. A full-bridge converter was chosen instead of a push-pull converter, because only one winding is used per primary (a split primary is needed for the push-pull) and the full-bridge will deliver a balanced waveform to the transformers.

The push-pull has only a single switch in the series path, compared to the two for the full-bridge converter. The push-pull is a better solution if the supply voltage is very low. The converter for the power supply operates from a 48 Vdc supply. The

additional voltage drop across the second MOSFET will not influence the efficiency or operation significantly and offsets the requirement for the split primary winding of the push-pull option.

The power supply has a main isolation transformer, with two additional transformers fed from the output of the main transformer, as shown in Figure 5-3. The secondary transformers must only provide isolation from the controller ground to the gate-drive circuit. If the controller ground reference is kept within the negative and positive DC-bus rails, the isolation need only cover the DC-bus operating range. This value will not change with additional levels added to the converter. The isolation rating of the main transformer must be able to withstand the voltage of the additional cells.



**Figure 5-3 – Transformer layout**

The photograph in Figure 5-4 shows the gate-drive interface and power supplies. The main isolation transformer can be seen in the center, with the two secondary transformers supplying the gate circuits closer to the edge.



**Figure 5-4 – Photograph of the gate interface and power supply**



## 5.4 Communications interface

### 5.4.1 Development of communications link

At the time of development there was no standard communications protocol that fulfilled the needs of the module controller. The only link designed for real-time applications was the SERCOS, but the need for high-voltage isolation and higher bandwidth and low cost prevented the use of any of the previously discussed standards. A custom standard was designed, based on the features from the standards mentioned in section 4.5.1.

Two different links were developed for the module controller. The first is based on a sampled, embedded clock link running at 10 Mb/s. The second link is based on the standard RS232 protocol transmitted via optic fiber.

The reasons for the inclusion of an RS232 link are threefold:

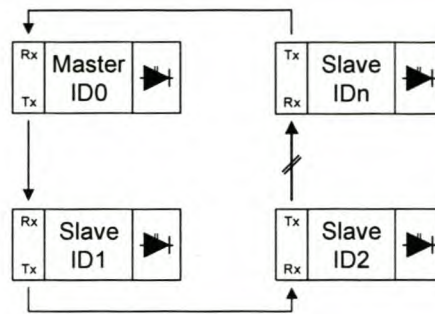
- Firstly to test the module controller with a minimal implementation of a communications link.
- The development software provided for the DSP used for controller is based on the RS232 protocol.
- RS232 allows the controller to easily interface with any personal computer with a serial port.

The design of the communications link follows the guidelines of the layered ISO network reference model, described in section 4.5.

#### Network and transport layer

As mentioned in the preceding chapters, the modules are connected in a ring configuration by daisy-chaining the output of one module to the input of the next module in the ring. Data flow on a single link is in one direction only (called simplex transmission), but the ring is a closed system, therefore data can be read from any device in the system. The interconnect configuration of the modules is shown in Figure 5-5.

Each module has a separate on-board clock signal, running asynchronously from the rest of the modules. The clock frequency is specified to be within 1% of the nominal clock frequency of 40 MHz.



**Figure 5-5 – Interconnecting the module controllers**

The link must be able to operate with a variable number of modules in the link. The total length of the link (bits stored) must be at least as long as the length of one packet. A packet is not allowed to circulate around the ring more than once. Only the main controller is allowed to transmit new packets on the link and is responsible for removing the previous packet after a successful trip around the ring.

Each module also contains a buffer to hold the data received and to keep data ready for transmission. The slave cannot initiate a transmit operation, but must be polled for data by the main controller. This remains true while the link is operating correctly. When the link is broken and a module detects that no data has been received from the preceding module, an alert packet can be sent (containing the module ID number) downstream to notify the main controller where the broken link was detected.

The clock frequencies of the devices are not synchronized; therefore the transmission of data runs plesiochronously (almost synchronous). This implies that a target can transmit data at a slower rate than it receives data from the previous device and a build-up of data can occur.

#### **5.4.1.1 RS232 optic fiber link**

The RS232 optic fiber link is used by the host software to communicate with the target devices. The configuration is shown in Figure 5-6.

The output of the serial port of the personal computer is converted to light by an optic fiber interface board. Only the data transmit and data receive pins are used. The link is then daisy-chained to a number of target devices, with the first device in the link designated as the master.



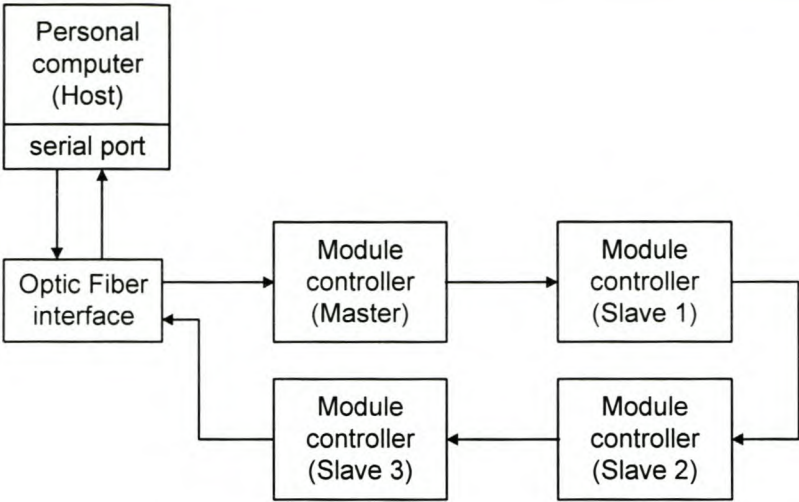


Figure 5-6 – RS232 optic fiber link

Data is received in frames with IrDA character stuffing to detect the beginning and end of a frame of data. There are two types of packets: (1) host-to-target packets and (2) target-to-host packets. The formats of the packets are shown below:

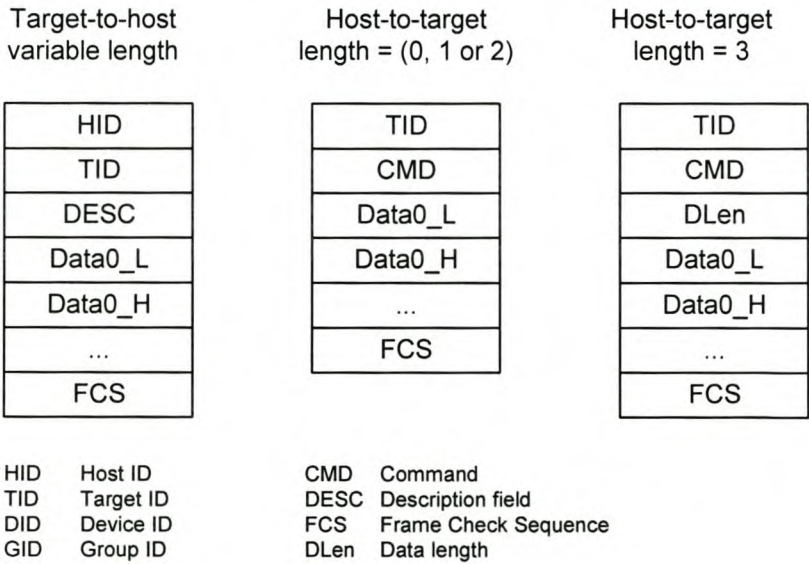


Figure 5-7 – RS232 packet formats

As the name implies, a host-to-target packet is used to send data to a target device. The packet included the target identifier (TID), command (CMD), data and frame check sequence (FCS). The target identifier consists of three fields, called the broadcast (BC), group identifier (GID) and device identifier (DID). They are used to determine if the target should respond to the packet. The command is also divided into three fields, called the read/write (R/W), length and command identifier (CMD\_ID) fields. The fields are shown in Figure 5-8.

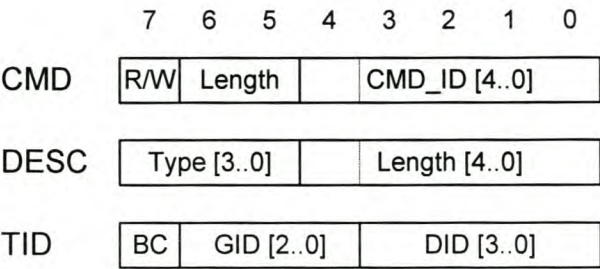


Figure 5-8 – Description of packet fields

*Addressing modes:* The host can send a frame to a specific target, broadcast frames to all devices within a group or broadcast the frame to all devices in the loop. If the BC bit is set, all devices will respond. Group broadcasts are done by setting the device ID to 0 (reserved for this purpose) and matching the group ID.

There is one exception to the rule and that is if the read/write (R/W) flag is set. A device will only respond to a broadcast if the read/write flag is cleared (write operation). This prevents multiple devices from flooding and jamming the communications link.

*Command:* The command consists of three fields, all of which must match for the command to be executed. The same command identifier (CMD\_ID) can thus be used by more than one command type by changing the length field and the R/W flag. For example, the commands for reading and writing a predefined reference will be the same, except for the R/W flag. The commands defined for use with the module controller are listed in Table 5-1.

Table 5-1 – Command definitions of module controller

Command	Description
CMD_NULL	NULL
CMD_SHUTDOWN	Shutdown device
CMD_START	Start operation
CMD_STOP	Stop operation
CMD_SLEEP	Enter sleep mode
CMD_WAKE	Wake from sleep
CMD_SYNC	Synchronize timing
CMD_RD_STATE	Read target state
CMD_RD_STAGE	Read target stage
CMD_RD_WDS	Read word from data space



CMD_RD_WPS	Read word from program space
CMD_RD_WIO	Read word from IO space
CMD_WR_WDS	Write word to data space
CMD_WR_WPS	Write word to program space
CMD_WR_WIO	Write word to IO space
CMD_RD_INFO_CNT	Read number of info lines
CMD_RD_INFO_STR	Read info line
CMD_RD_WADS	Read word array from data space
CMD_WR_WADS	Write word array to data space
CMD_RD_REF	Read reference
CMD_WR_REF	Write reference
CMD_WR_REF_BOUNDS	Write reference with bound check
CMD_RD_MEAS	Read measurement
CMD_DACQ_SINGLE	Perform single data acquisition
CMD_DACQ_SELECT	Select channels for acquisition
CMD_SET_TIME	Set target time
CMD_CALC_CRC	Calculate CRC of block

*Target-to-host:* When a command is sent with the R/W flag set (read operation), the master/host expects a reply from the target module. The format of the return packet is specified in Figure 5-6. The header defines the length of the payload data and the type of data. The packet is addressed to the host by using the identifier reserved for the host (HID). The packet also contains the target identifier. This enables the host to track the source of the received data.

The data type is used by the presentation layer of the host software to translate the specific data format of the target to the data format of the host computer. The predefined formats are listed in Table 5-2. The TMS320F240 uses 16 bits to store the signed and unsigned integers. The floating-point format is a 32-bit IEEE float. These are translated by the host application to the internal format used by the personal computer.

Table 5-2 – Predefined data formats

Type	description
DATA_TYPE_CONTROL	Control codes
DATA_TYPE_INT_ARRAY	Integer array
DATA_TYPE_UINT_ARRAY	Unsigned integer array
DATA_TYPE_CHAR_ARRAY	Character array
DATA_TYPE_FLOAT_ARRAY	Float array
DATA_TYPE_REJECT	Reject command

5.5 Main control board

The main control board consists of a digital and an analog section. The digital section consists of the DSP (Digital Signal Processor), memory, PLDs (Programmable Logic Device) and miscellaneous drivers and interfaces. The analog section consists mainly of power supply measurement circuitry and comparators used for hysteresis control and over-current protection.

5.5.1 Digital section

The digital section forms the heart of the controller and is responsible for executing the control code and handling all of the communications with the outside world.

The section consists of the optic fiber transmitters and receivers, the two EPLDs, the DSP, SRAM and interface buffers. The block diagram is shown in Figure 5-9.

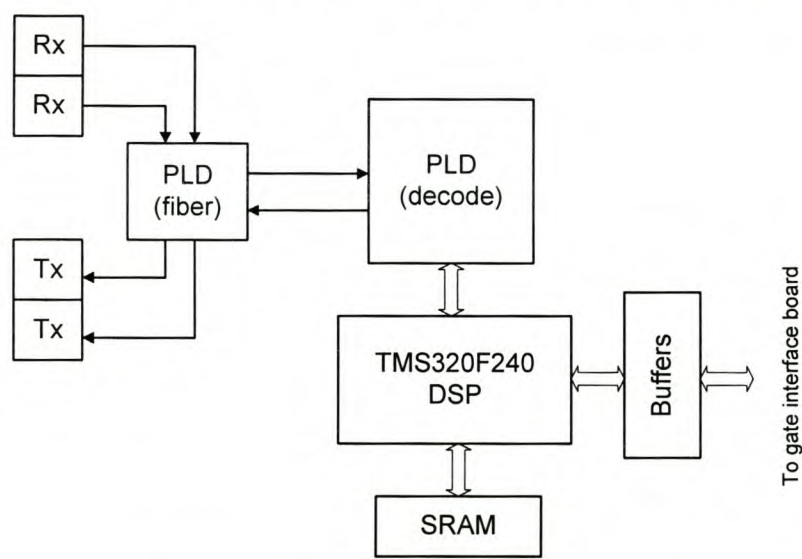


Figure 5-9 – Digital controller block diagram



*Processor:* The processor is a TMS320F240 DSP (Digital Signal Processor) targeted specifically at embedded systems [48]. The device has internal FLASH memory for program code, allowing the device to be reprogrammed in the field. The DSP has two ADCs (Analog to Digital Converters) on-chip, each with an eight input multiplexer and a resolution of 10 bits. This allows a total of 16 channels to be measured.

*Programmable logic:* Two EPLDs (Erasable Programmable Logic Devices) supply the resources for implementing the external logic needed for address decoding and the high-speed communications link. The EPLDs also combine the error signals generated by the different protection circuits to drive the power protect interrupt of the DSP. The errors are stored in a dedicated error status register for later analysis. The power drive protect interrupt (PDPINT) of the DSP is a hardware-level signal for disabling the gate-drive signals routed to the gate-drive board. This is the preferred option because it bypasses the software and has a minimal latency from error to trip.

The block diagrams of the EPLD firmware are shown in Figure 5-10 and Figure 5-11. There are slight variations between the master and slave devices, mainly in the generation and detection of carrier and reference synchronization pulses.

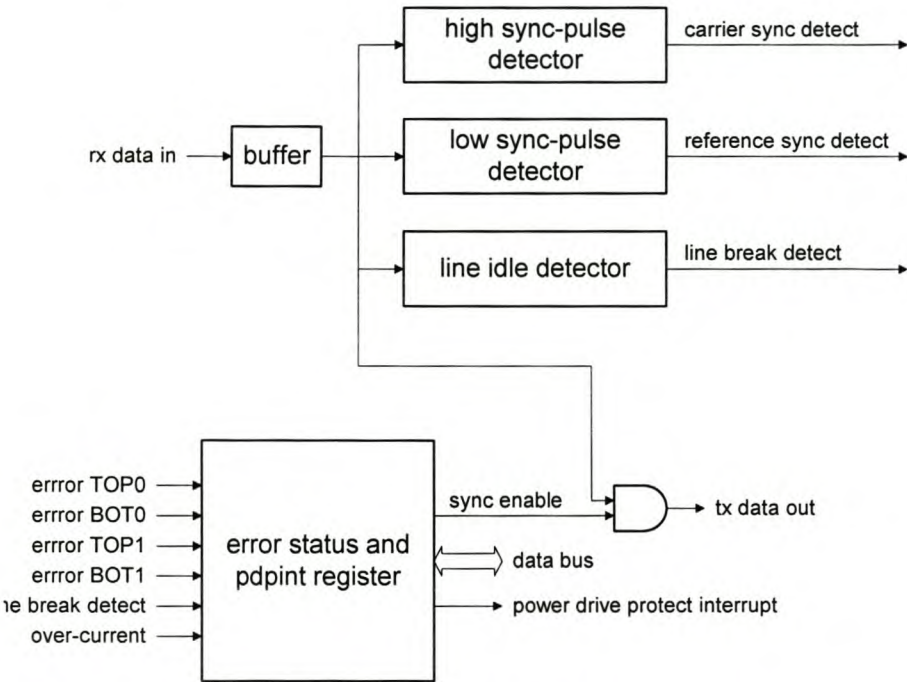


Figure 5-10 – Block diagram of EPLD firmware (slave)

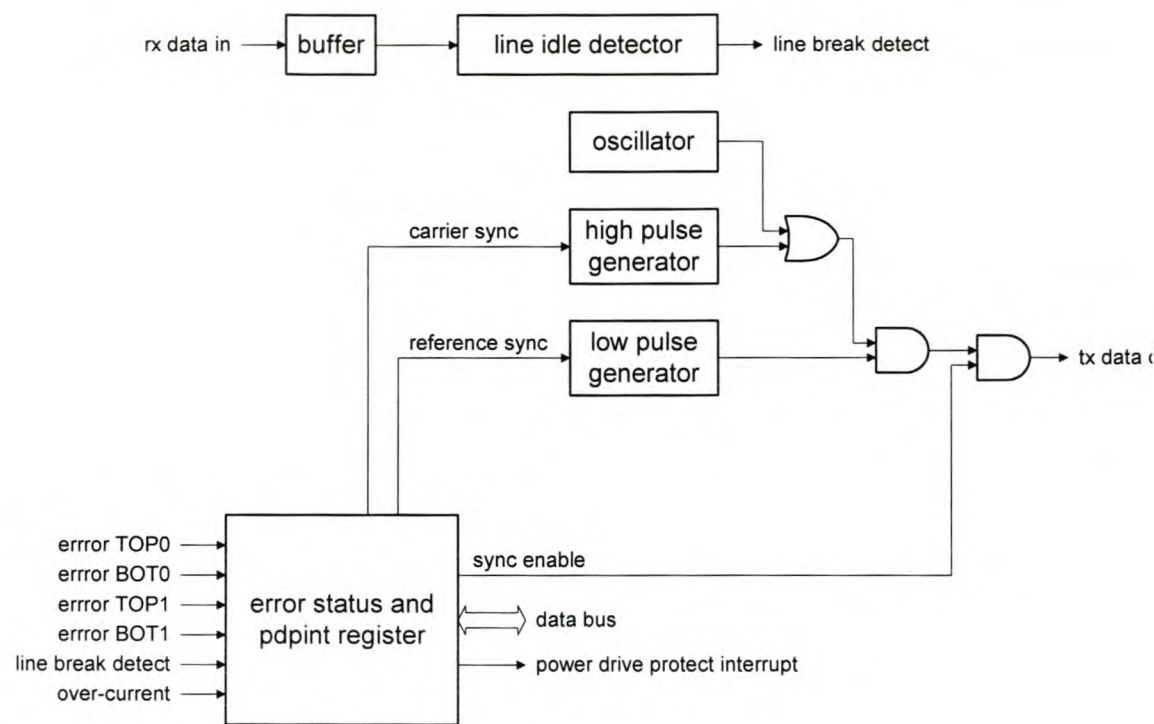


Figure 5-11 – Block diagram of EPLD firmware (master)

Unfortunately, the logic resources are not sufficient to completely implement the high-speed optic-fiber link together with the rest of the support circuitry needed for the operation of the module controller. Only sections of the high-speed data extraction could be implemented and tested. The final evaluation version of the module controller implemented a synchronizing link only and together with the RS232 link provided for communication and low-level synchronization and protection of the system.

*Memory:* The slave modules are supplied with 32 kW of external SRAM (static ram). The memory is used to store waveform data captured by the data acquisition unit coded into the embedded software. The data can be retrieved online by the host software. All program code executes from internal FLASH memory of the DSP controller. The F240 is a 16-bit DSP and uses a paged memory scheme to extend the limited address range of the DSP. The data, program and input/output pages overlap the same address range, but are differentiated by means of three separate strobe signals.

The internal FLASH memory is mapped into the program space only (which is difficult to access from c-code); therefore it is advantageous to copy data from the



internal FLASH memory to the external SRAM before executing the program code. Each module has an embedded string array for storing information concerning the hardware and software capabilities of the slave device. An intelligent master can use this information table to dynamically configure the slave devices during the start-up sequence.

*Optic fiber transmitters and receivers:* The module controller has two sets of optic fiber transmitters and receivers. The first set is used for the RS232 optic fiber link implementation. The optic transmitter and receiver are a low-cost plastic fiber solution from Agilent. The receiver has a built-in quantization circuit and can operate at speeds of up to 5 MBd. The devices are very simple to use and no additional circuitry is needed to interface with the fiber receiver. The driver for the optic fiber transmitter is based on a quad nand gate. The outputs are connected in parallel to supply the current needed for the LED in the fiber transmitter.

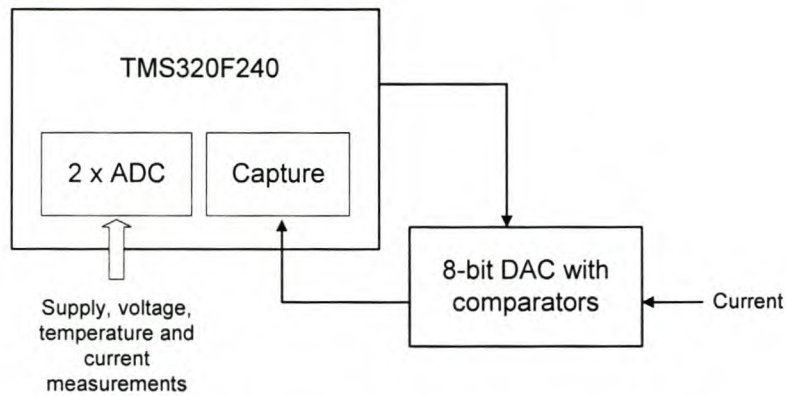
The second set of fiber transmitter and receiver is based on a higher-speed plastic fiber solution from Agilent. The transmitter is a higher-efficiency version of the above-mentioned transmitter and used the same driving circuit. The receiver does not contain a built-in quantization circuit. The quantizer is based on a high-speed differential comparator circuit. The input signal has a varying DC component (temperature dependent) and is therefore coupled to the quantizer circuit via a high-pass filter with a low cut-off frequency (similar to the AC-coupled version). The dynamic signal range of the received voltage is relatively small (typically 0.7 V maximum deviation) and the circuit is therefore more sensitive to noise. The circuit theoretically allows for speeds of 0 to 32 MBd, according to [39]. The practical evaluation of the circuit is shown in section 6.2.

*Buffers and drivers:* The rest of the circuit consist mainly of the necessary buffer and driver needed to interface with external devices. The buffers that drive the signals to the gate-drive board are open-collector drivers, allowing more than one voltage level to be used. The input of the gate-drive board is the diode of an opto-coupler and together they form a simple current loop. This type of interface is less sensitive to noise coupled from external sources.



### 5.5.2 Analog section

The analog section consists of the DSP internal analog to digital converters, the power supply measurement unit, two temperature sensors and an external DAC with built-in comparators. The measurement unit is defined as a separate entity and is discussed in the next section. The block diagram is shown in Figure 5-12.



**Figure 5-12 – Block diagram of controller analog section**

*Temperature measurements:* The controller board of the module controller has an on-board temperature measurement sensor for measuring the ambient temperature of the controller. The temperature can be used to detect over-heating conditions, as this will adversely affect the long-term operation of the controller.

An external temperature sensor can also be connected to the board to measure the heatsink temperature. This temperature measurement can be used to detect over-heating of the power switches. Most of the third-generation intelligent IGBT modules incorporate some form of thermal trip. This should only be allowed as a last resort, because only a limited number of trips are allowed during the lifetime of the device, caused by warping of the module base plate, etc.

The temperature measurement can also be used to dynamically control the switching frequency of the converter. If the temperature of the converter is too high and it is imperative that operation not be interrupted, the switching frequency can be decreased to lower the switching losses of the system. This scenario is typical of a UPS (uninterruptible supply) type application.

*Supply voltage measurements:* The module uses a maximum of five supplies. At least three supplies are needed for operation of the device; they are the +5 V and  $\pm 15$  V supplies. The two additional supply lines are unregulated  $\pm 18$  V supplies to



optionally supply current to the active hall-effect sensor used for current measurement (LEM LA 205-S 300A current sensor). The regulators for the primary supplies are located on the controller board. The input and the output of each of the supplies can be measured by the DSP controller and can be used for protection and online monitoring.

*Supply current measurements:* The primary supplies are fed through current shunts before entering the voltage regulators. The voltage across the shunt is measured by a differential amplifier and passed to the DSP for measurement. The measurements can be used to detect fault conditions and provide online status information.

*External DAC:* The controllers are supplied with a quad 8-bit DAC (digital to analog converter) with built-in comparators. The DACs can be used to set trip levels that are monitored by hardware. Two of the DAC channels are fed by the converter output current measurement, allowing the module controller to use a form of hysteresis current control. The outputs of the comparators are connected to the capture inputs of the DSP. This allows the DSP to measure the time the converter spent in a certain state to a high degree of accuracy (50 ns resolution).

Figure 5-13 and Figure 5-14 show photographs of the top and bottom views of the module controller main board. The top view shows the DSP, SRAM clock and drive circuits. The bottom view shows the two EPLDs and the optic fiber transmitters and receivers.

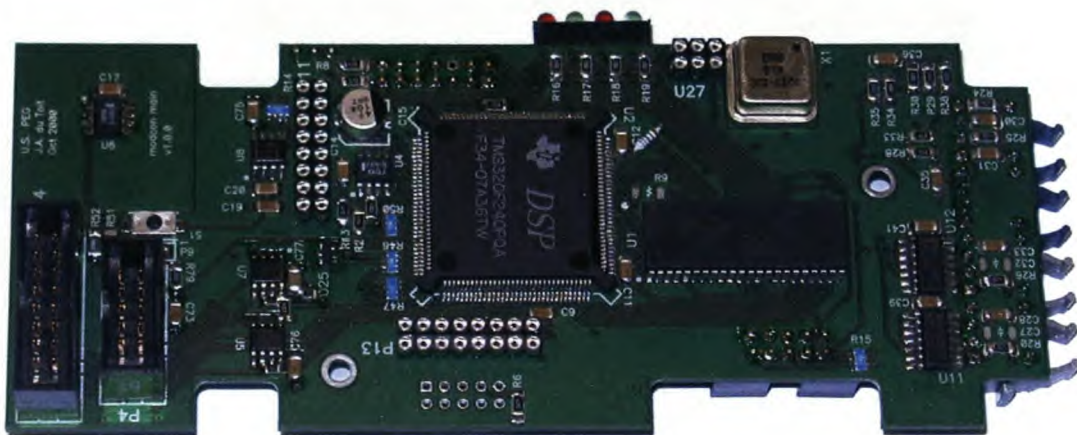


Figure 5-13 – Top view of module controller main board

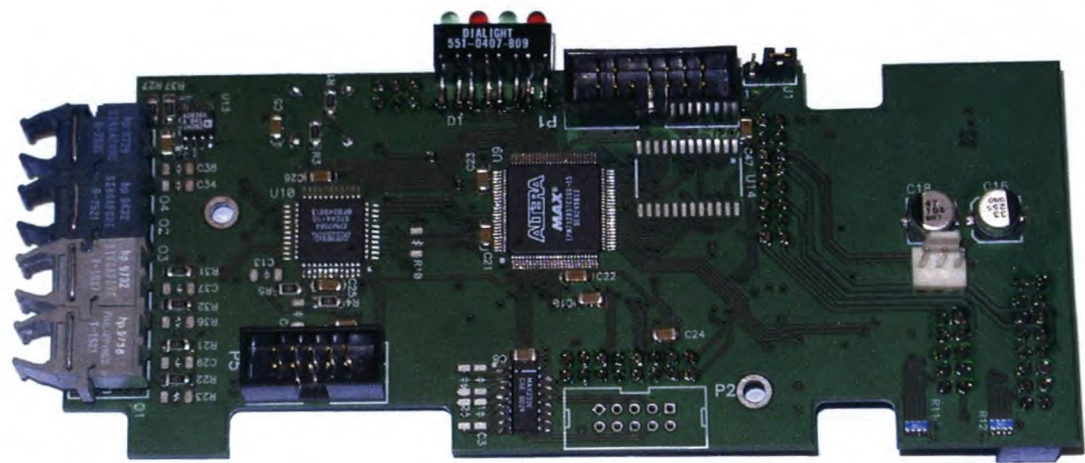


Figure 5-14 – Bottom view of module controller main board

5.6 Measurement board

The measurement board is used to gather local parameters and system variables (e.g. currents and voltages of the converter and supply/load) for control and protection purposes. The individual measurements on the measurement board are described in the following sections. The block diagram of the measurement unit is shown in Figure 5-15.

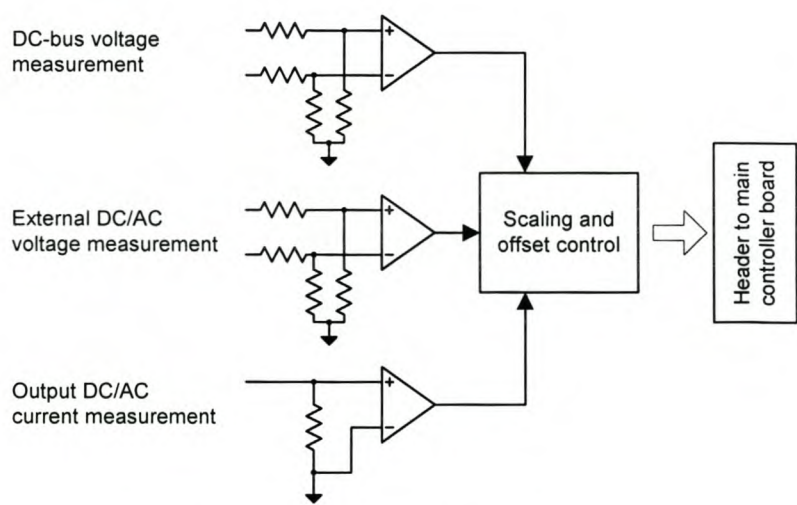


Figure 5-15 – Block diagram of measurement unit

Figure 5-16 shows a photograph of the top view of the measurement unit. The bottom of the board contains the circuitry of the power supply current measurements.



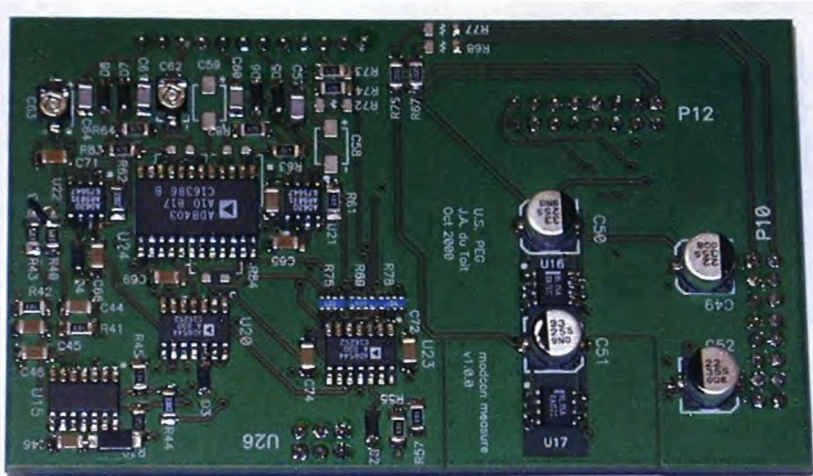


Figure 5-16 – Photograph of measurement unit

5.6.1 Voltage measurement

The controller is capable of measuring two differential voltage measurements. The first measurement is dedicated to measuring the DC-bus voltage. This is an important measurement, because an over-voltage will lead to the destruction of the power switches. The measurement uses a differential input, allowing the system ground of the controller to be referenced to any point within the range of the DC-bus upper and lower limits. The controller ground can therefore be connected to the positive DC-bus, the negative DC-bus or the DC-bus capacitor center tap, as shown in Figure 5-17. The reference points for the controller ground are marked as GND ref GT, GC and GN respectively (short for ground top, ground center and ground bottom). The gate-drive circuit ground potentials are marked as DRV ref ET and EB (drive emitter top and ground emitter bottom).

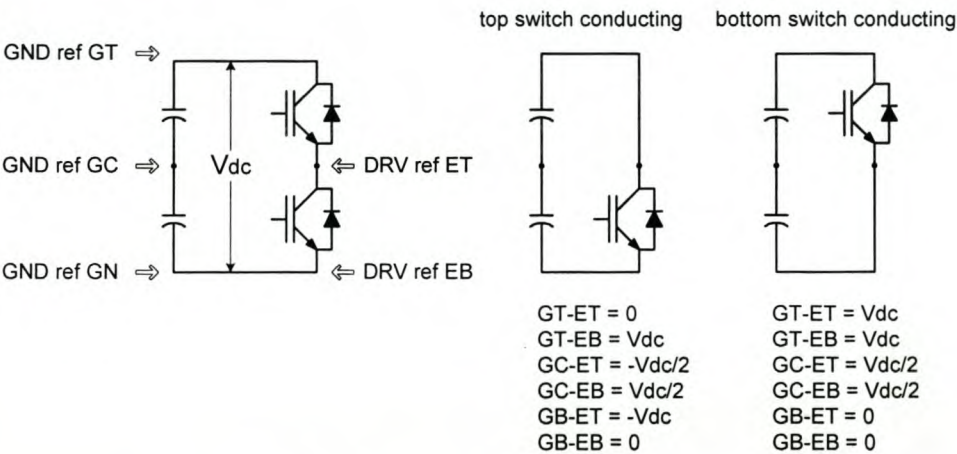


Figure 5-17 – Ground reference points for controller



Depending on the state of the top IGBT, the potential of ET will be close to the negative or positive DC bus (GB and GT respectively). If the module controller ground is referenced to either the top or bottom DC-bus voltage, the isolation circuits of the gate-drives must be able to handle at least the full DC-bus voltage. With the ground referenced at the capacitor center tap (GC), the maximum deviation is half the DC-bus voltage and the slew rate is thus also halved. The requirements for the gate-drive power supply and opto-couplers can therefore be reduced.

The second voltage measurement (identical to the first, except for an increased common mode range) can be used to measure either the output of the converter or the DC-bus capacitor center tap voltage. The specific requirements for the measurements were discussed in section 3.4.

### 5.6.2 Current measurement

The current measurement is used to measure the output current of the converter. The module controller implementation is based on an active Hall-effect current sensor that can measure from DC to 100 kHz. The current can also be measured with a current transformer (CT), but this allows only for the measurement of AC currents. A third approach is to use a current shunt at the output of the converter. The small differential voltage across the resistive shunt is usually associated with a large common mode voltage and high slew rate caused by the switching of the converter. This makes it difficult to measure the current accurately.

The current can also be estimated by measuring the saturation voltage of the power switches. This method is currently in use in a number of devices, although the measurements are only used for over-current protection and not regulation. The integrated measurement would be the ideal solution, because the external components add to the cost of the module. In the case of the Hall-effect current sensor, the sensor must also be supplied with power from the local isolated power supply.

## 5.7 Protection of power module and controller

This section gives a summary of all the protection that is available on the module. Some of the protection was specifically added to prevent damage to the system caused by mistakes made during development and testing of the system and might not be included in a final industrial design.



The protection options are:

- Power switch error (generated by the IGBT driver)
- Converter over-current (hardware trip level)
- DC-bus over-voltage (implemented with software)
- Heatsink temperature measurement (software test)
- Power supply voltage measurements
- Power supply current measurements
- External analog mono-stable based watchdog
- Controller ambient temperature measurement
- RS232 communications watchdog timer
- Carrier synchronization pulse watchdog (generated by hardware)
- Reference synchronization watchdog
- DSP watchdog timer
- Software stage synchronization
- Software address validation (generated internally by the DSP)

The protection falls into four broad categories:

*Converter power stage:* The operation of the converter power stage is protected from over-current, over-temperature, short-circuit and over-voltage conditions. The controller supplies the gating signals for the IGBT. To prevent shoot-through, the controller inserts blanking time between the top and bottom signals. Any error signal from the module will immediately cause the gating signals to become inactive. The DSP is supplied with a watchdog timer to protect the devices if some external source or software bug causes the software to malfunction. An analog watchdog was added to the IGBT gate-drive board to ensure that the device enters an idle state if a trigger pulse does not change level within a specified period. This will also ensure that the drive board will shut down if the clock fails or the connection between the controller and the driver board is broken.

*Controller and power supply:* The power supply protection includes the current and voltage measurements of the individual supplies. They can be used to detect over-current situations and verify that the secondary circuits are powered. The controller ambient temperature can be used to monitor the temperature during certain periods of activity. If the ambient temperature becomes too high, the components might start to



fail and could lead to erroneous gating pulses. The temperature will also influence the calibration of the measurement system.

*Software:* This category protects against mistakes made during the development of the control software, but can also be used for the final product for built-in self-test functionality. This category includes the DSP watchdog timer, software address validation and software stage synchronization. The real-time nature of the controller software requires that certain operations be performed on a regular basis. The stage synchronization will detect if the execution time of a subroutine exceeds the maximum allowable time for that stage.

*Communications:* The loss of the communications link during operation can cause the system to become unstable. To prevent this unsafe situation, both of the communications links are monitored for activity. If the idle time exceeds the specified period, the converter will be shut down. The idle detection of the RS232 is only applicable when the modules are in an active running state.

## **5.8 Module synchronization**

Not every situation and application requires that the modules should be synchronized at the PWM carrier frequency. Topologies like the stacked converter require that the PWM carriers be synchronized if the natural balancing of the converter is used.

The modules of the ring are synchronized using a solution that requires both hardware and software. The clock frequency is specified to be within 1% of the nominal 40 MHz. The DSP uses a divide-by-two clock for internal operation. Switching at 5 kHz results in 4000 clock cycles per switching period. The maximum deviation is therefore 40 clock periods from the nominal 4000. Practical verification verified that all the clocks were within 0.01% of the nominal 40 MHz.

The solution for synchronizing the modules requires both hardware and software. The carrier sync pulse detected by the EPLD is fed to the capture input of the DSP. A change on the pin captures the current value of timer 2 and stores this value. Timer 2 is also the counter that is used for the stage timing. The captured value is then compared to a reference value to determine if the local clock is running too slow or too fast. The frequency of the carrier and stage counters are then modified to lock onto the phase and therefore the frequency of the carrier. The frequency of the carrier is adjusted by modifying the maximum value of the ramp counter.



The master controller generates the carrier sync pulses. The sync pulses are usually embedded within the data of the high-speed optic fiber link. Unfortunately, the programmable logic resources available on the module controller were insufficient to implement a complete high-speed communications link. A very simple communications link was implemented, capable of generating and detecting reference and carrier synchronization pulses. The slave devices detect the synchronization pulses and generate the appropriate interrupt and capture pulses for the DSP. A hardware watchdog timer in the EPLD detects idle time on the communications link and will notify the DSP of this situation. The idle time can be caused by a faulty fiber or forced by controller if a critical error condition occurs on the module. All of the devices in the link can then be shut down simultaneously.

Converter topologies like the cascade converter typically switch at the fundamental frequency of the power system. A dedicated reference synchronization character or command can be used to synchronize the 50 or 60 Hz reference using the RS232 link. If a high-speed communications link is not available, the AC power supply can also be used to generate synchronization pulses.

## **5.9 Controller software**

The embedded software must perform the following functions:

- Converter control (e.g. current and voltage regulation)
- Protection (software current limits, etc.)
- Data acquisition (waveform capture)
- Real-time communication with host software

The software listing for the module controller is given in Appendix C.

The software can be broken down into two major categories:

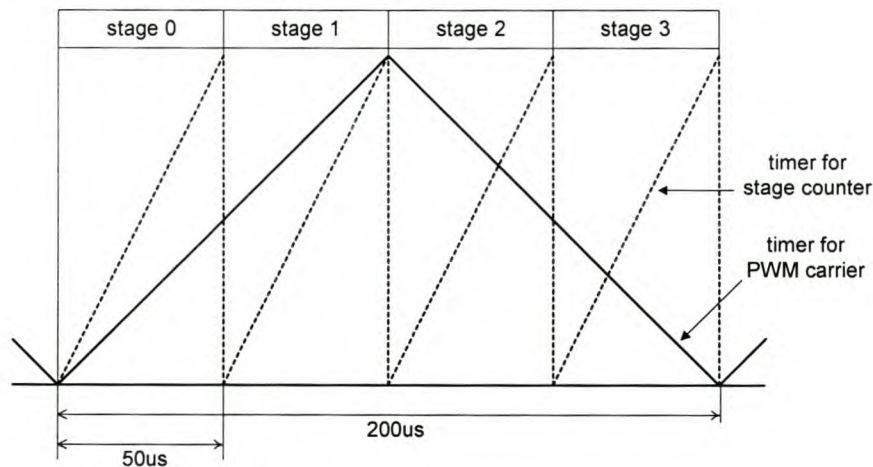
- Code that must execute at a specific time (synchronous time critical code).
- Code that can run asynchronously.

The module controller is set to switch at a fixed frequency of 5 kHz, but certain operations must be performed at a higher rate, for example, the measurement and communications sub-systems. The timing of the software is critical and is discussed in more detail in the next section.



### 5.9.1 Module controller software timing

This section describes the timing of the module controller. The controller makes use of two of the internal timers of the F240 for pulse width modulation and stage timing. DSP timer 1 is used for the PWM carrier and runs at half the DSP clock frequency. The timer is set to operate as an up/down counter, resulting in a symmetrical or asymmetrical PWM signal, depending on the update rate of the PWM references. The references are double-buffered and can be updated at the end of stage 1 and stage 3. DSP timer 2 is used for the stage counter and runs at the DSP clock frequency. The timing relationship of the various ramp timers is shown in Figure 5-18.



**Figure 5-18 – Internal timing of the controller**

An interrupt is generated at the start of each new stage. The values are then read from the analog to digital converters and stored in the ADC tables for later use. The values that are read from the ADCs were converted at the end of execution of the previous stage. The conversion of a new set of measurements is then started. This set is used for the time-critical measurements (DC-bus voltage, output voltage and output current).

While the ADCs are busy converting the next set of measurements, the stage-specific code is executed and the serial communications interface is serviced. After the completion of the analog to digital conversion, the next set of measurements is started. This set includes the less time-critical measurements like the temperature and power-supply voltage measurements. Depending on the stage number, the new PWM reference value is calculated and written to the PWM double-buffers. After each step



the stage time is saved for debug and self-test purposes. The progress of the stage execution is shown in Figure 5-19.

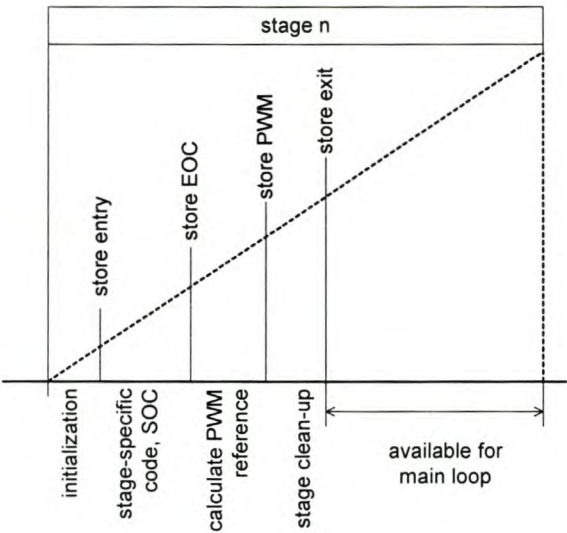


Figure 5-19 – Controller stage timing

Only the time critical-code is executed within the interrupt routine. New data from the communications channels are stored in buffers for later evaluation. After the interrupt routine exits, the remainder of the 50  $\mu$ s period is available for executing the asynchronous main control and supervisory code.

It is important that controller execute the correct code for each stage number. If the code of one stage exceeds the limit of 50  $\mu$ s, synchronization will be lost. The timing can be checked once every switching period. This is done during stage 0 execution. The value and counting direction of the PWM carrier counter is read and checked (must be smaller than half of the maximum counter value and counting up).

There are two additional stages defined that are used during initialization of the controller. These are used for locking onto the incoming carrier synchronization pulses and to ensure that the stages are synchronized to the PWM carrier (four stage periods per PWM carrier period).

5.9.2 Operational states of the controller

Depending on external events (host or master commands, hardware interrupts etc.), the state of the controller can change to reflect the new operational mode. The pre-defined stages are listed in Table 5-3.

Table 5-3 – Operational states of the controller

State name	State definition
OPSTATE_UNDEF	Undefined state, error condition
OPSTATE_NORM_RUN	Normal with outputs enabled
OPSTATE_NORM_IDLE	Normal idle, outputs disabled
OPSTATE_TIMEOUT_IDLE	Timeout occurred on communications link
OPSTATE_SYNC_LOST	Timeout occurred on sync-pulse extract circuit
OPSTATE_SHUTDOWN	Module is shut down
OPSTATE_SLEEP	Module enters sleep mode
OPSTATE_TIFLSH	Module idle, ignores commands from host
OPSTATE_WAIT_SYNC_LOCK	Module waiting for synchronization lock
OPSTATE_WAIT_STAGE_SYNC	Module waiting for stage synchronization

Under normal conditions, the states will occur in the following order:

- (1) OPSTATE\_WAIT\_SYNC\_LOCK (waiting for sync pulses)
- (2) OPSTATE\_WAIT\_STAGE\_SYNC (waiting for stage sync)
- (3) OPSTATE\_NORM\_IDLE (module ready and waiting for start command)
- (4) OPSTATE\_NORM\_RUN (module running)

The rest of the states are only entered after an unexpected event or a request from the host software.

5.9.3 Program flow

The flow of the program code is summarized using the flow-charts shown in Figure 5-20 and Figure 5-21.

Figure 5-20 shows the flow of code for the main asynchronous loop, as well as the interrupt entry and exit code. The main loop is responsible for the initialization of the controller and peripherals, after which it services the higher levels of the communications interface. There are two interrupt routines, each with multiple interrupt sources. The interrupt routine verifies that a legal source caused the interrupt before continuing with the code. Figure 5-21 shows the stage specific code executed within the interrupt routine.



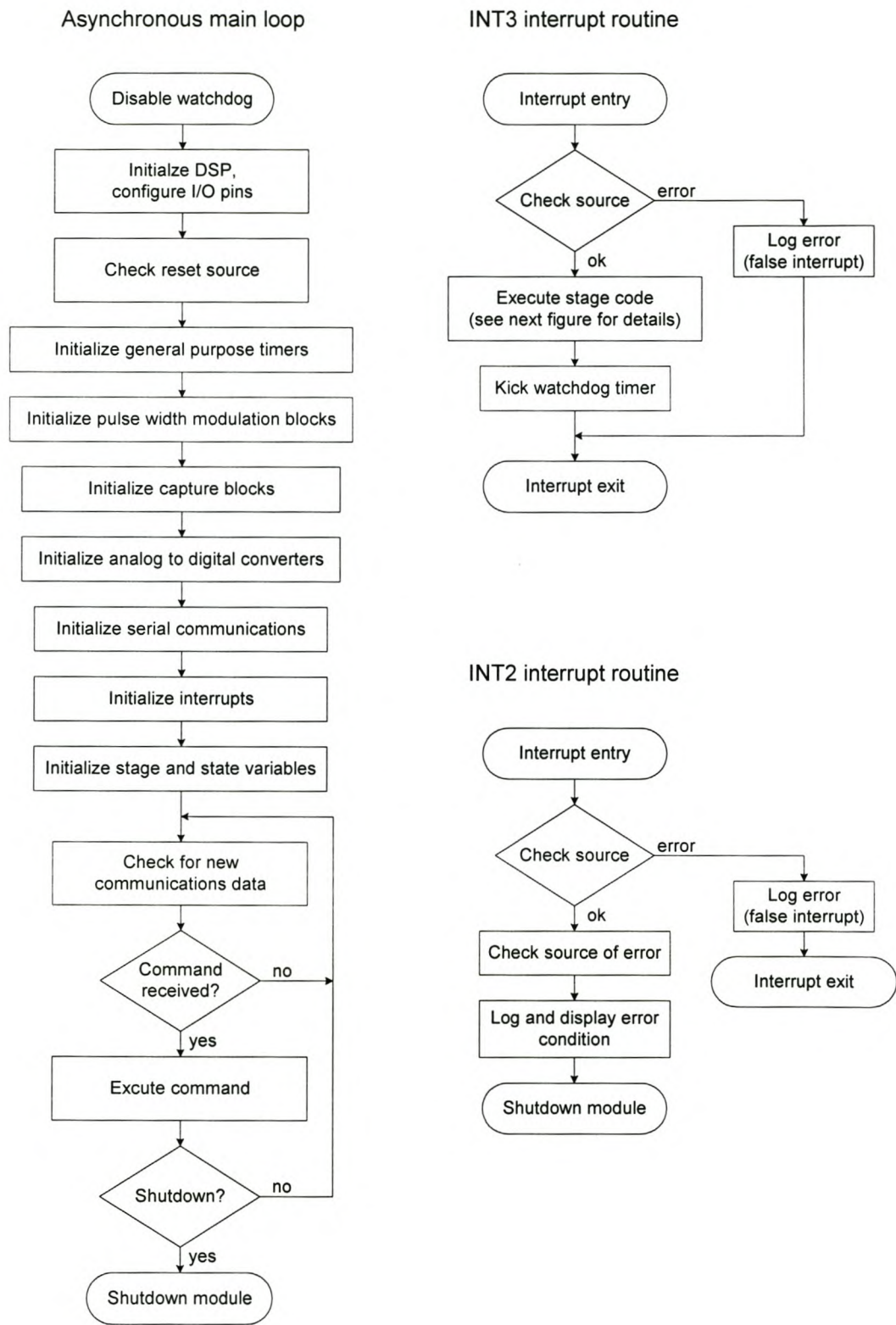


Figure 5-20 – Flow-chart of embedded code (main and interrupt entry/exit)

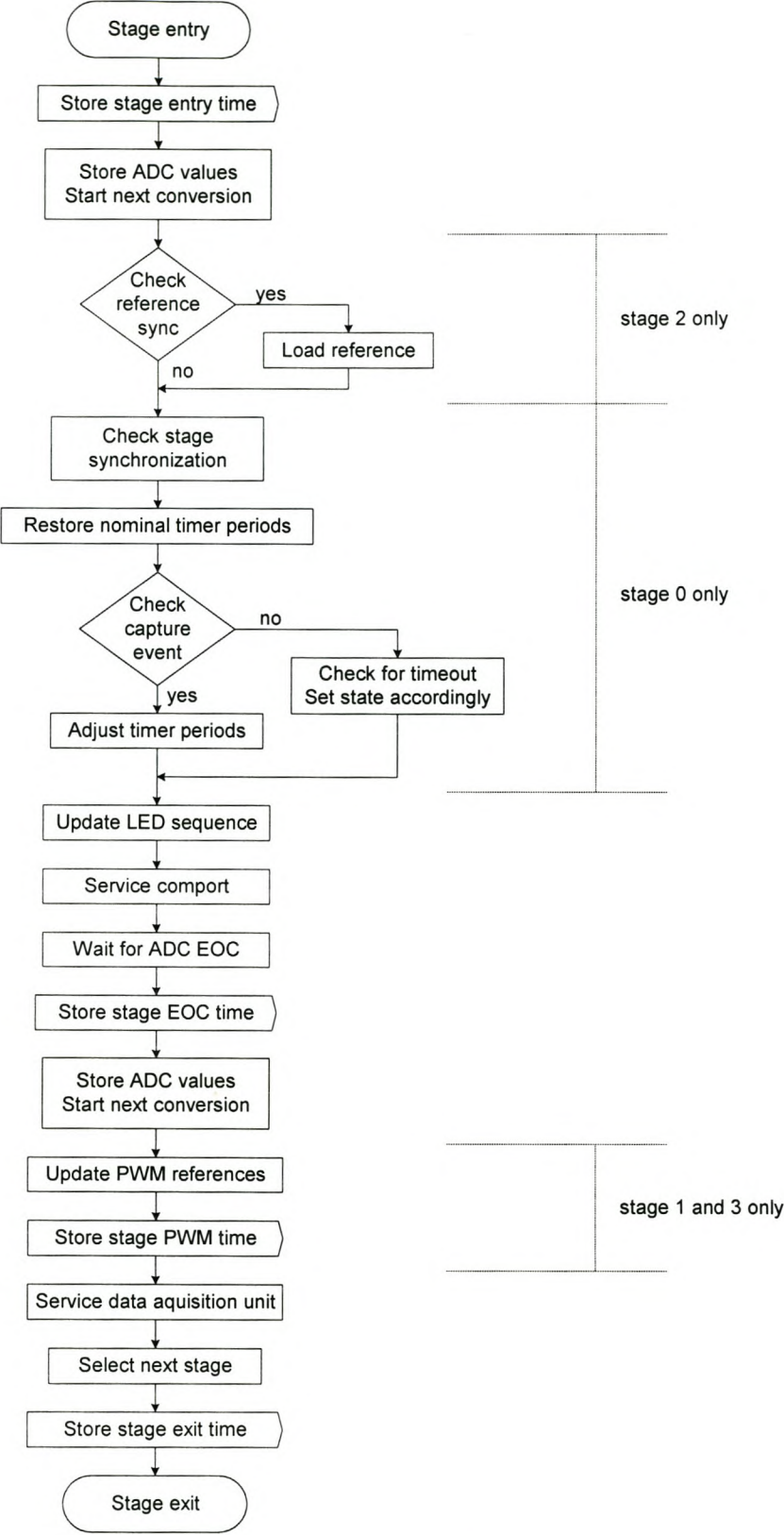


Figure 5-21 – Flow-chart of embedded code (stage specific)



### 5.10 Host Interface Software

This section gives a short description of the host controller software running on a personal computer. The PC side programming is of secondary importance for the dissertation and a detailed description of the software will therefore not be given. This section is here to describe the flow of data from the visual and control components to the target controllers.

The software takes a layered approach, enabling the main application to use different physical interfaces (for example, the RS232 fiber-optic and high-speed serial interfaces) and devices on multiple communication links.

All classed derived for the client and server applications are fully polymorphic, with base classed defined to allow run-time initialization and instancing of objects. All communication with the target devices is through a server application. This allows multiple client programs to communicate simultaneously with the target devices.

#### 5.10.1 Data flow through layers

The flow of data from the source to the destination is best described visually, as shown in Figure 5-22. The layers were specifically designed so that each consecutive layer does not need to have knowledge of the type of information contained in the packet.

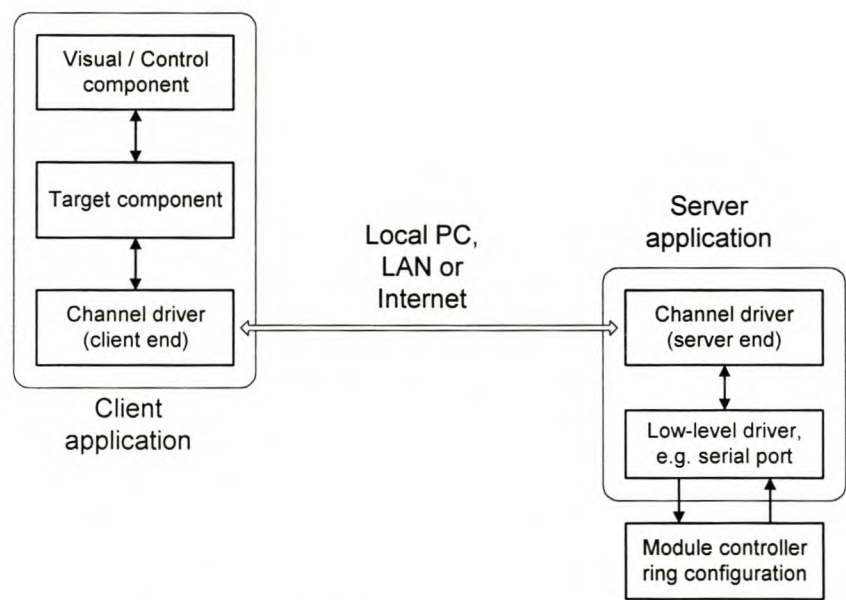


Figure 5-22 – Data flow through the host software

Each of the components in the layered architecture is implemented as a C++ class. Below is a brief description of each class in the hierarchy, with a description of the main function that each class performs.

A target is defined as any final destination for a data packet. Targets include a module controller or any system capable of receiving and interpreting data compatible with the module controller interface and protocol. An isolated measurement unit can be designed based on the module controller protocol. Data can then be exchanged freely between the master and the measurement unit during operation.

#### *5.10.1.1 Visual and control components*

The visual and control components are derived from a class (TClientObj) that forms the base for all client objects. Components derived from TClientObj can send requests and receive data. The component sends a command by directing the call to an associated TTargetObj derivative (discussed in the next section). Any data transmitted in response to the command are routed back to the TClientObj derivative.

Some of the components derived from this class are the graph/waveform display components, the target information display component and the measurement/variable display components.

#### *5.10.1.2 Target component*

All target components are derived from a base class called TTargetObj. The TTargetObj base class exposes functions to send packets of data to the specific physical target. The TModCon component is derived from this class and is responsible for assembling the packets in the correct format for decoding by the module controller. The TModCon component is associated with a TComChannelObj component for sending the packet to the server application.

The target component takes the command sent from the visual or control component and formats the data for interpretation by the physical target. The component thus has knowledge of the target hardware (size of integers and floats, memory map, etc.). The packet is assembled and the information added to a packet info structure. The structure is then passed to the next layer, in this case the channel layer.



### 5.10.1.3 Channel components

The channel components form a pair, one for the client side and the other for the server side. The client side receives data from multiple client objects and passes the data through to the channel driver component to the server side.

The TComChannelObj derivative components do not have or need knowledge of the type and format of the data contained within the packet. This allows the channel to connect to multiple, different targets. The interpretation of the data is left to the target components.

All server-side channel components are derived from one or more base-classes, depending on the type of server. Different base classes can be developed for each type of connection, for example, sockets for Internet and named pipes for local area networks. The derived class then adds the functions needed to communicate with the physical targets.

The current implementation of the software makes use of named pipes to connect the client and server-side components. The class is named ModCon232 service and communicates with the module controllers through the RS232 serial port. The server can handle connections from multiple clients simultaneously.

The main server application is implemented as an object that can be embedded into either a standard server application or a windows service application. Service applications are notoriously difficult to debug and security issues make this an even more daunting task. For this reason the server object was written to allow embedding into either a standard windows application or a service application.

The final message that is assembled and sent to the server is shown in Figure 5-23. The status window of the server application is shown in Figure 5-24. The window shows the connection status for the four named pipe connections allowed for the server. The statistics of each pipe are monitored individually and displayed. The server keeps track of the number of packets sent and received on each pipe and the number of errors encountered. The window also shows the status of the serial port. The same information is displayed for the serial port as for the named pipes.

Target index Channel index Packet index Client * Target * Client timeout Flags	Message information
Reserved	Target-specific information
Server command Packet ID Server timeout Flags	Channel-specific information
Target to host packet, Host to target packet	Packet data

Figure 5-23 – Final message structure

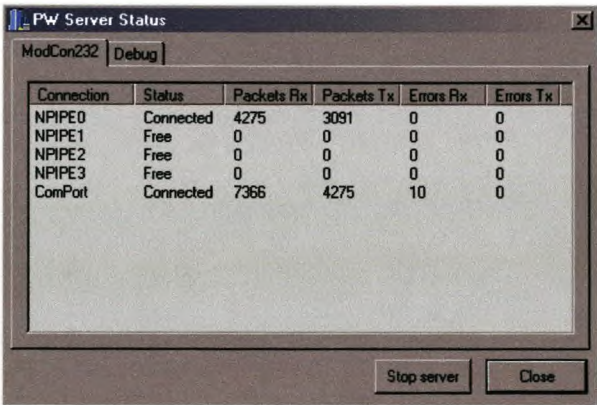


Figure 5-24 – Server status window

The client window is shown in Figure 5-25. The client windows show the graph component, a module controller control component, a target information component and a generic command test window. Multiple clients can access the same server and module controller simultaneously. Commands generated within the same client application are given a priority class (low, normal and high) and queued for transmission to the server.

The target information component retrieves target specific and host defined variables from the target. The variables are in a readable text format and gives information on the capabilities, etc. of the target device. This information can be used for initializing and auto-configuration of the chain.



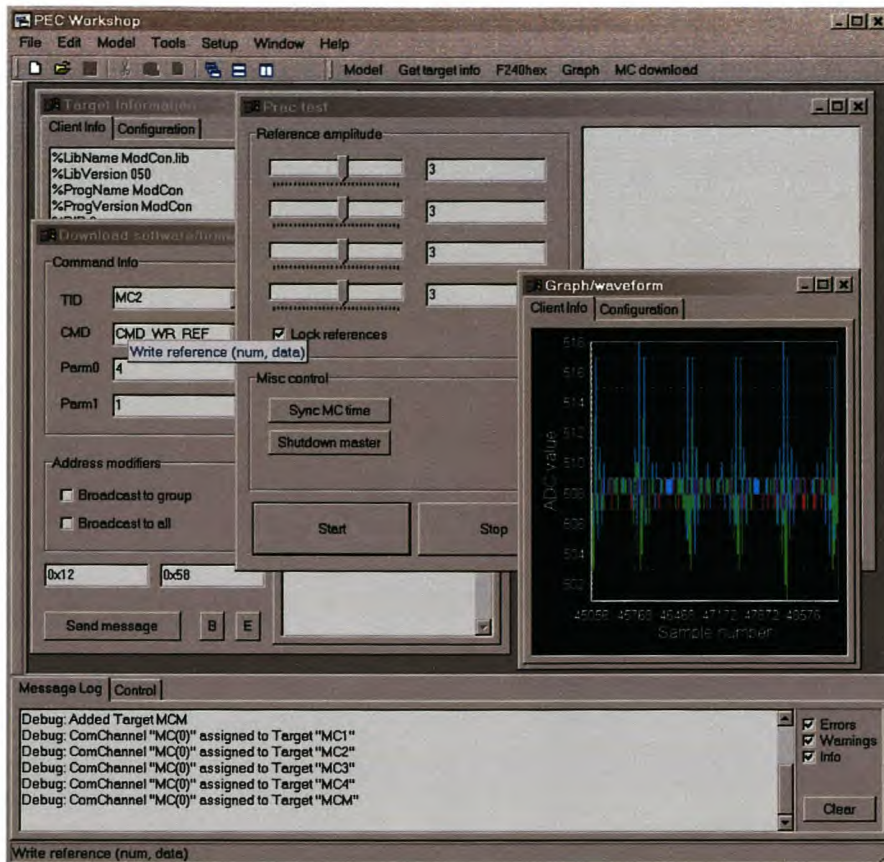


Figure 5-25 – Client application window

## 5.11 Summary

The chapter described the design of a module controller used for evaluating the distributed control of power electronic converters. The design specifically targeted the development phase of the product and control strategies.

The chapter discussed the design of individual components of the system, ranging from the hardware needed for the controller and power-supply to the software needed for control and debugging.

From a protection point of view, the module controller should handle nearly any fault condition that occurs within the system. This might make the system hard to test, because the controller will trip on any spurious event.

Development is handled by communicating with the controllers through a client/server application developed specifically for this project. The reason is that there was no software available for interacting with the target system, but this also gave the chance to fully exploit and evaluate the layered approach needed for the open-ended interaction needed for a fully industrialized or commercial application.

# Chapter 6

---

## Evaluation of the distributed controller



## Chapter 6 Evaluation of the distributed controller

### 6.1 Introduction

The evaluation and experimental results are divided into three main sections. The first section deals with the operation of the individual components of the module controller.

The second section deals with the evaluation of the module controller in the various configurations described in section 3.4. This section therefore tests the module controller as a stand-alone power block.

The third section deals with the evaluation of the module controller within a distributed environment. The section therefore evaluates the choices made during the design of the interfaces and topology evaluation.

A photograph of the system is shown in Figure 6-1. The stack has four slave module controllers and an additional master controller (shown on the bottom shelf).



**Figure 6-1 – Photograph showing converter stack**

## 6.2 Evaluation of module controller subsystems

### 6.2.1 Optic fiber quantization circuit

The operation of the high-speed fiber quantization circuit is shown in Figure 6-2. The bottom trace is the output signal of the fiber receiver. The fiber optic connector was lifted slightly from the receptacle to lower the amplitude of the received signal. The peak-peak value of the received signal is approximately 200 mV. The inverted output of the quantization circuit is shown in the top trace. At this low level of power, there is only slight distortion in the received signal. The distortion can be seen by the uneven pulse duration of the positive and negative pulses.

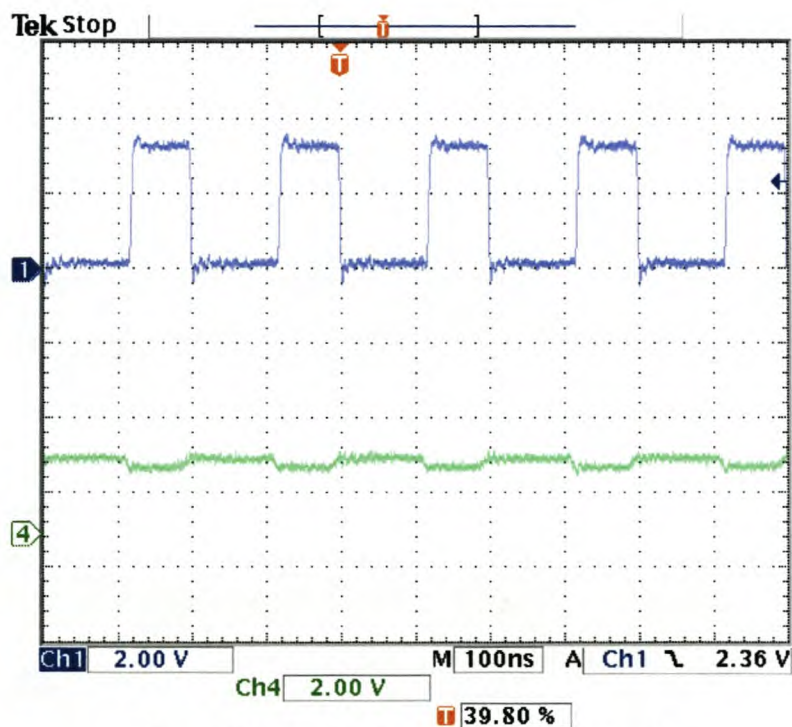


Figure 6-2 – Operation of quantization circuit

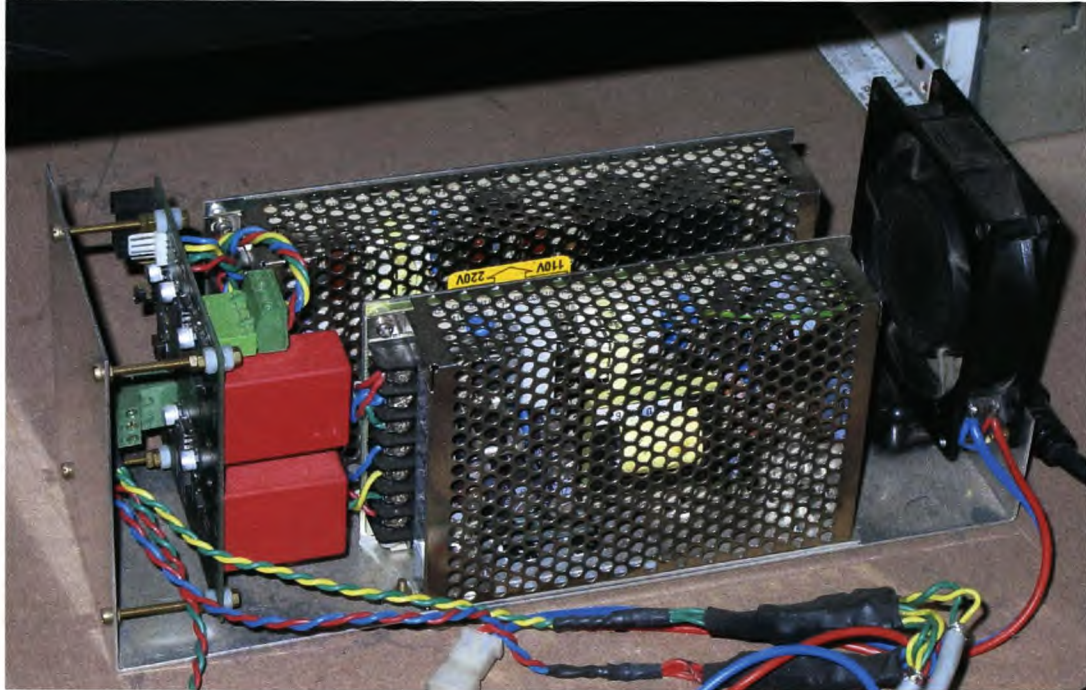
### 6.2.2 Distributed AC power supply

Figure 6-3 shows a photograph of the main power bridge of the distributed AC supply. The two supplies on the right provide the regulated 48 V for the main bridge and the secondary +5 V and +15 V for the control circuitry.

The supply was evaluated at low load (single device) and full load to test the performance at both extremes. The switching noise was also evaluated by examining the effect of the switching on the most sensitive components on the controller board. The analog stages are more susceptible to noise from the supply than their digital



counterpart. The measurement system and fiber optic receivers are the only analog circuits on the board. The instrumentation amplifiers used for the voltage measurements have a high supply noise rejection value and can therefore be ignored. Other measurements like the power supply voltages and temperature measurements have filters to minimize the effect of noise.



**Figure 6-3 – Photograph of the distributed AC supply**

The only real cause for concern is the fiber optic receiver. The receiver used for the high-speed link does not have a built-in quantizer. A circuit using a high-speed differential analog comparator with hysteresis is used to quantize the analog signal from the fiber optic receiver. The input of the circuit is a high-pass filter; therefore the circuit is more sensitive to noise when the link is idle.

The switching waveform of the power supply full-bridge is shown in Figure 6-4. The switching frequency of the bridge is approximately 270 kHz with a peak-to-peak voltage of 104 V at the output. The voltage after the isolation transformer has a peak-to-peak amplitude of 21 V. This value is sufficient for the 15 V regulators.

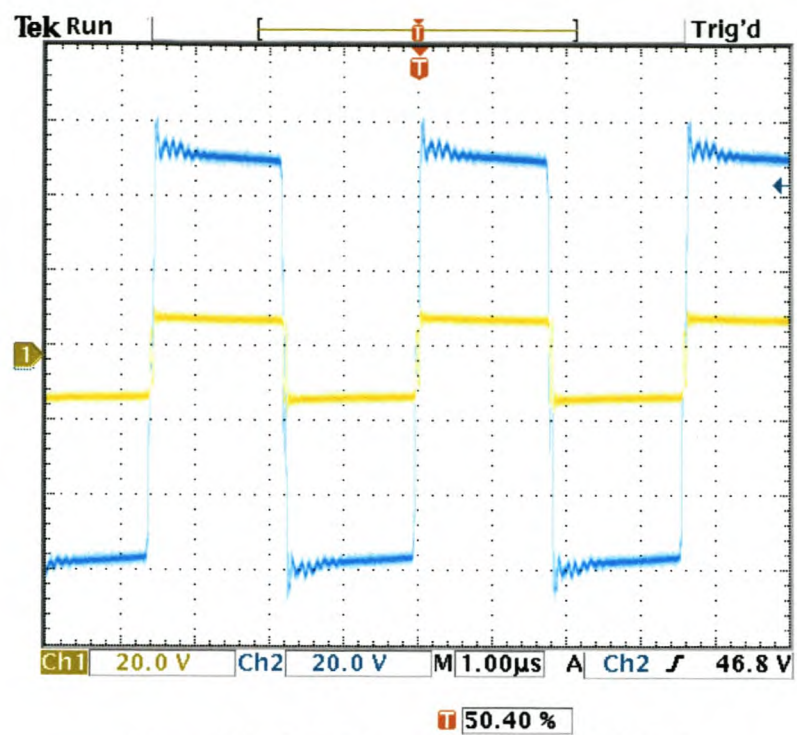


Figure 6-4 – Switching waveform under full load

Figure 6-5 and Figure 6-6 show detail of the switching behaviour for light and full load respectively. The transient has an adverse effect on the rectified voltage and influences the minimum size of the filter components.

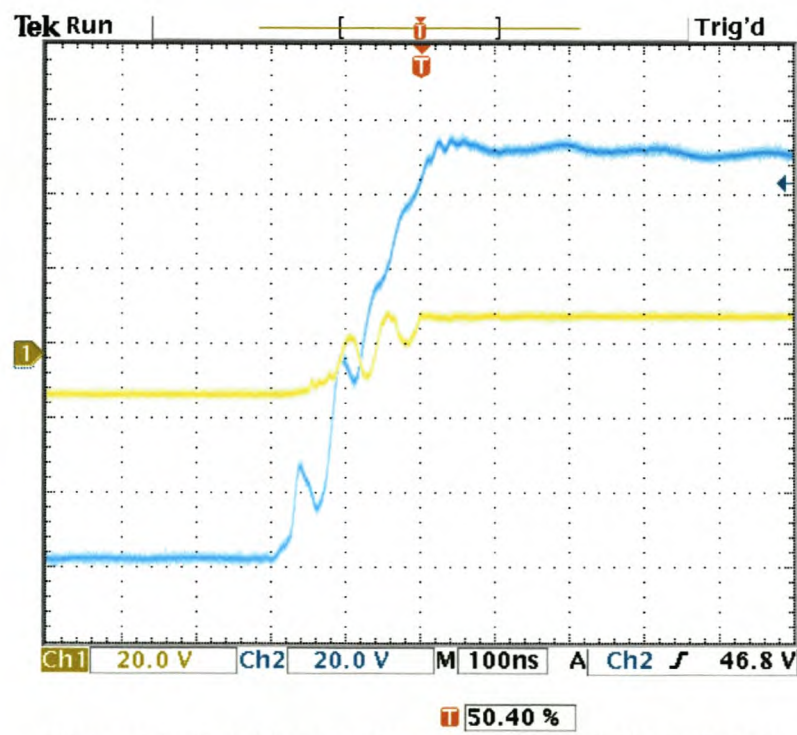


Figure 6-5 – Switching transient under light load (single device)



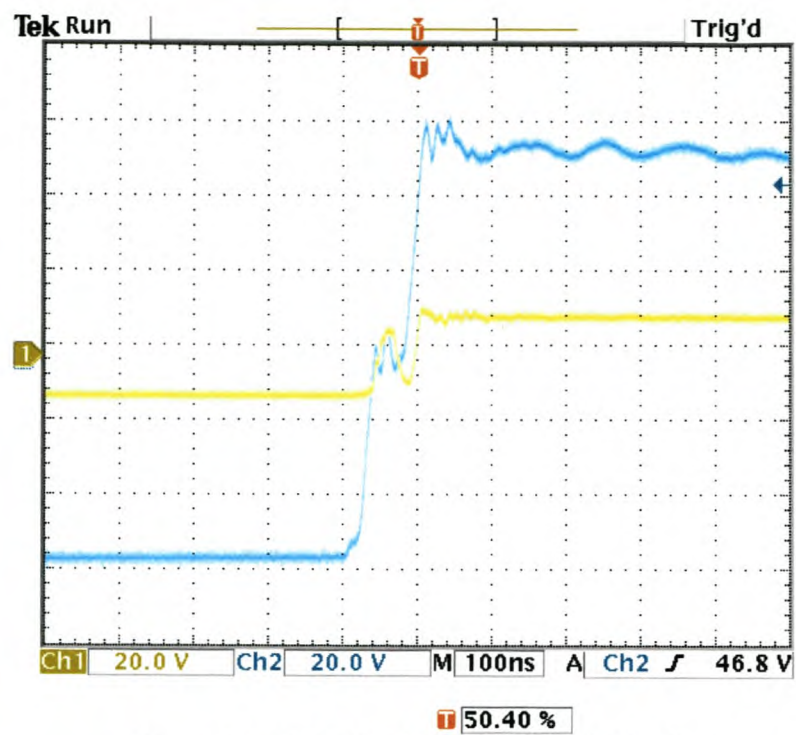


Figure 6-6 – Switching transient under full load

The noise sensitivity test is shown in Figure 6-7. Trace (1) shows the noise present on the 5V supply. Trace (2) is the output of the quantizer with the fiber removed from the input. It should be noted that the measured signals are influenced by the oscilloscope and probes used for the measurement, as both the oscilloscope and measured system have floating ground potentials.

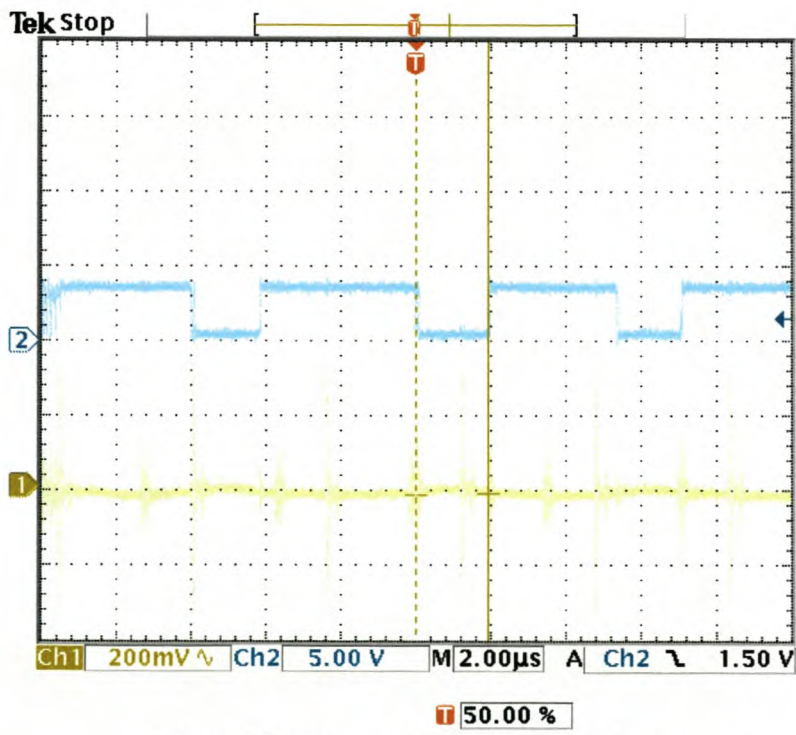


Figure 6-7 – Output of fiber optic quantizer

6.2.3 Protection

The protection of the module controller falls into one of 3 categories:

- Communications fault
- Power device fault
- Synchronization/control

The protection was tested for each of the cases mentioned above. For all cases the controller should do a controlled shutdown and force the rest of the distributed system to enter a valid operational state.

The reaction of the system to fault condition is demonstrated in Figure 6-8. The graph shows the PWM output of the modules. Trace number 4 is the trip signal (IGBT error feedback) and trace 1 is the PWM output of the module that experienced the error. The response time is almost immediate. Traces 2 and 3 show the output of modules lower down the link. The time it takes for the modules to trip is approximately 2  $\mu$ s.

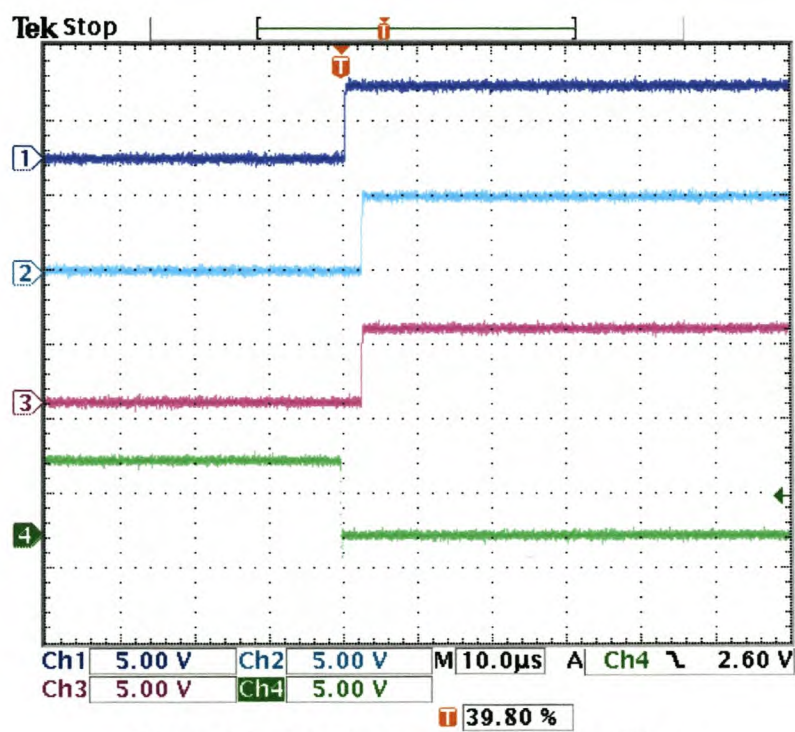


Figure 6-8 – Reaction time after a trip condition



6.2.4 Data acquisition

Each module controller has support for data acquisition using the on-chip ADCs and external memory for storage. To verify the operation of the acquisition unit, the module controller was configured as a current regulator. The currents before and after the filter capacitors were measured using an oscilloscope and are shown in Figure 6-10 and Figure 6-9 respectively. The current before the filter capacitors contains all of the ripple caused by the switching of the converter.

Depending on the current regulator, one of the two measurements can be used in the algorithm for the current regulation. For sampled systems (typical of digital implementations), the current is sampled at least once within each switching period. By examining the current waveform measured before the filter capacitors, it can be seen that even a small deviation in the sample time will cause a large offset in the sampled value of the current.

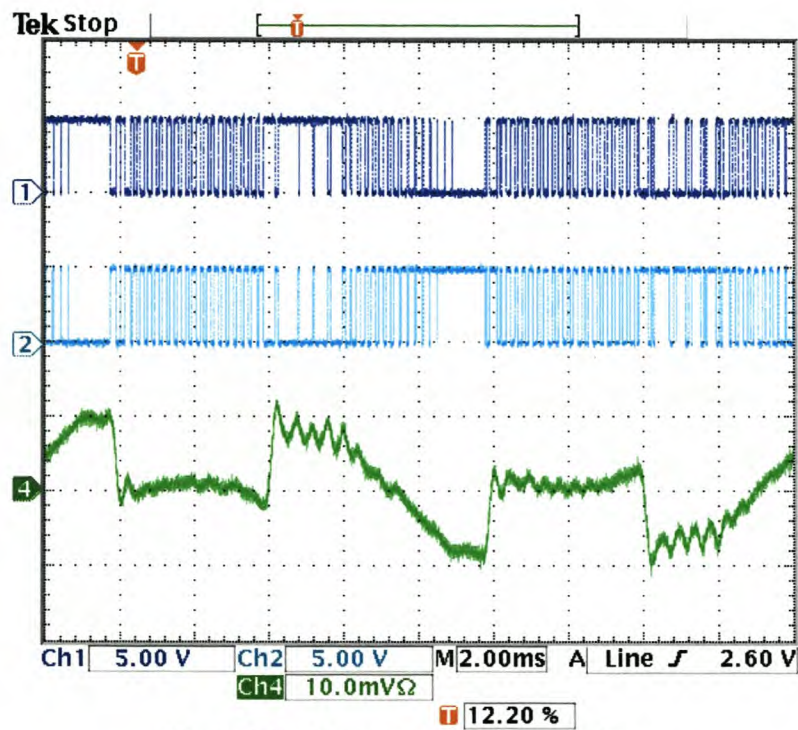


Figure 6-9 – Current after filter capacitors

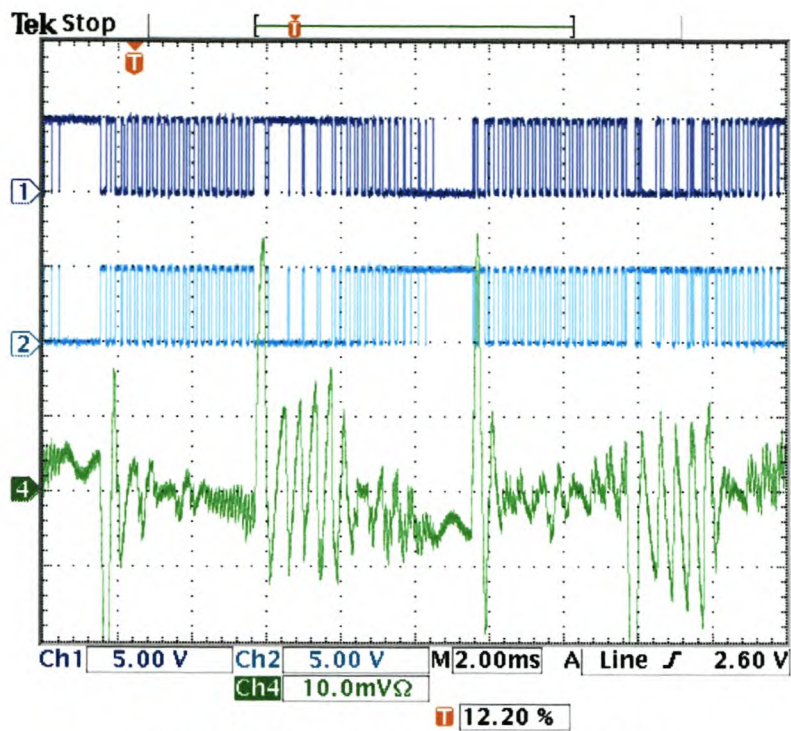


Figure 6-10 – Current before filter capacitors

The sampling instants of the data acquisition unit are shown in Figure 6-11. The output current is stored at each sampling instant (50  $\mu$ s period) and the two voltage measurements are interleaved (100  $\mu$ s period).

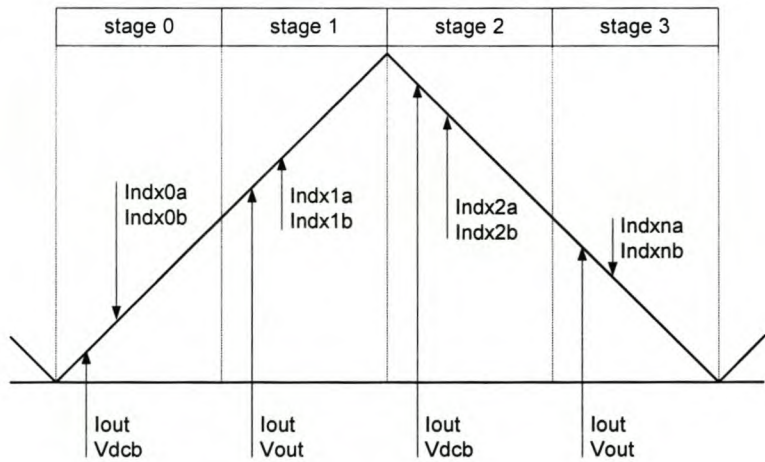


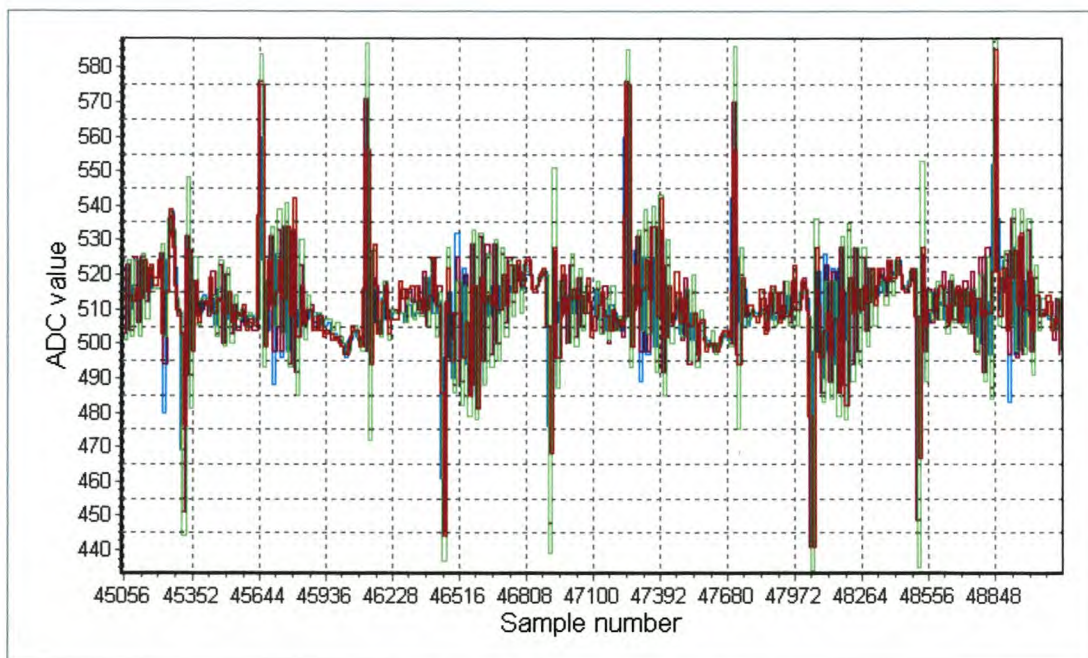
Figure 6-11 – Sampling instants of data acquisition unit

The module controller was used to measure the current before the filter capacitor. The same current was used for the current regulator. The results are shown in Figure 6-12. The horizontal axis is the sample index and the vertical axis is the ADC value. The data is sampled at 20 kHz and the ADC has a resolution of 10 bits (unsigned value). A value of 512 therefore corresponds to a current of 0A.



The graph shows four traces. Each trace corresponds to a specific sampling time within the switching period. Traditionally only one sample per switching period would be used for the current regulation algorithm (typical of homogeneously sampled PWM). The four traces therefore show the different current waveforms that the controller would acquire for each of the sampling times.

The order of sampling is (0) red, (1) cyan, (2) purple and (3) green. The sampling instances are  $50\text{ }\mu\text{s}$  apart and spread evenly over the switching period. The conversion start is initiated from software and calculation time prevents the first sampling instant from being zero. This would have been the ideal case, as the sampling instant falls within the zero state, almost independent of the pulse width.



**Figure 6-12 – Current sampled by module controller**

### 6.2.5 Synchronization

The low level synchronization is necessary for a number of the converter topologies. This also allows optimal control of ripple by making explicit use of the interaction between phases and converters.

The generation and detection of the sync pulses are shown in Figure 6-13. The reference sync pulse is the inverted form of the carrier sync pulse.

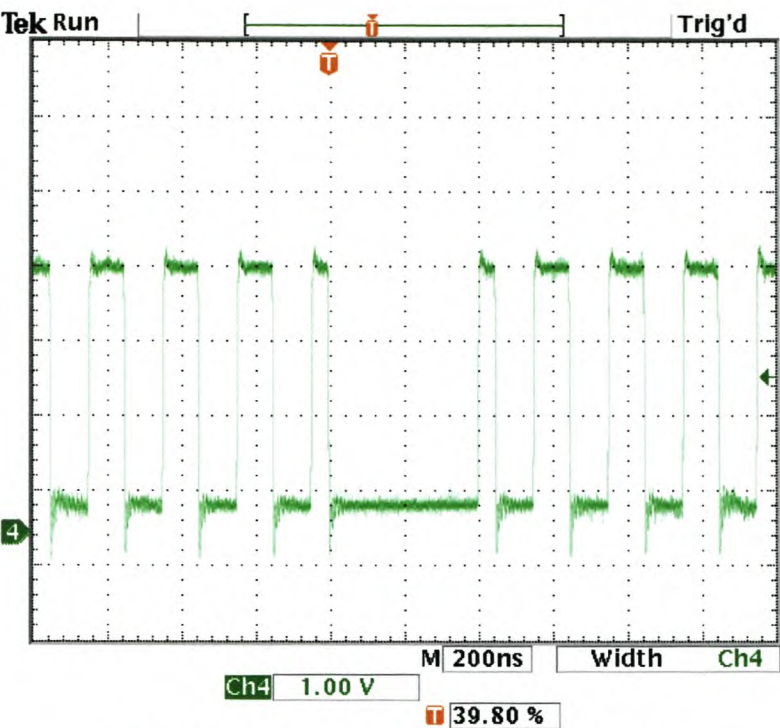


Figure 6-13 – Carrier sync pulse received by slave

The performance of the phase lock loop used for the carrier synchronization is shown in Figure 6-14. The display was set with a persistence of 10 s. The waveform shows the maximum deviation of the carriers within that period.

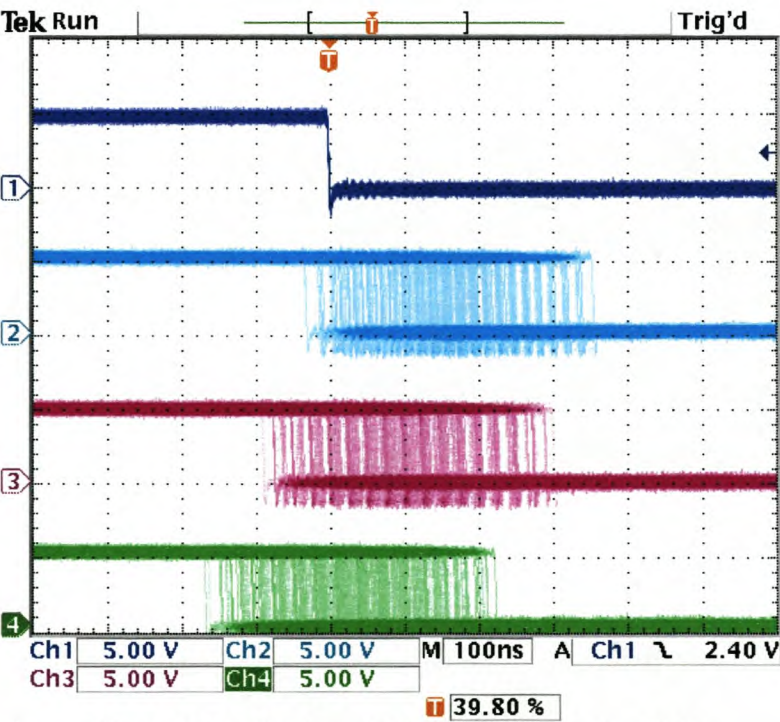


Figure 6-14 – Synchronization of modules using PLL



The maximum deviation is measured at approximately 300 ns, which is 0.15%. It can be noted that each successive device has a delay of about 75 ns. This is a predictable delay through the EPLD synchronizing circuit. The clock frequency of the fiber EPLD is 40 MHz, resulting in an average of three clock cycles delay through the EPLD and receiver/transmitter circuit.

### 6.3 Evaluation of module with standard configurations

The module controller can be configured to operate in one of four modes. A single module controller is used for the tests.

#### 6.3.1 Current regulator

The module controller is used to regulate a reference current generated internally. The setup for the evaluation is shown in Figure 6-15.

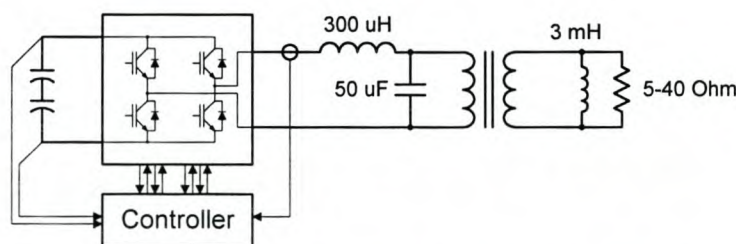


Figure 6-15 – Test setup for current regulator configuration

The load is a linear load consisting of a parallel connection of an inductor and a resistor. The result for a sinusoidal reference current is given in Figure 6-16. The DC-bus voltage is 450 V and the scale is 50 A per division.

The current measured by the module controller is shown in Figure 6-17. The four sampling instants are shown as individual traces. The measurements sampled during the zero-states have noticeably less ripple and gives a better approximation of the average current for the switching cycle.

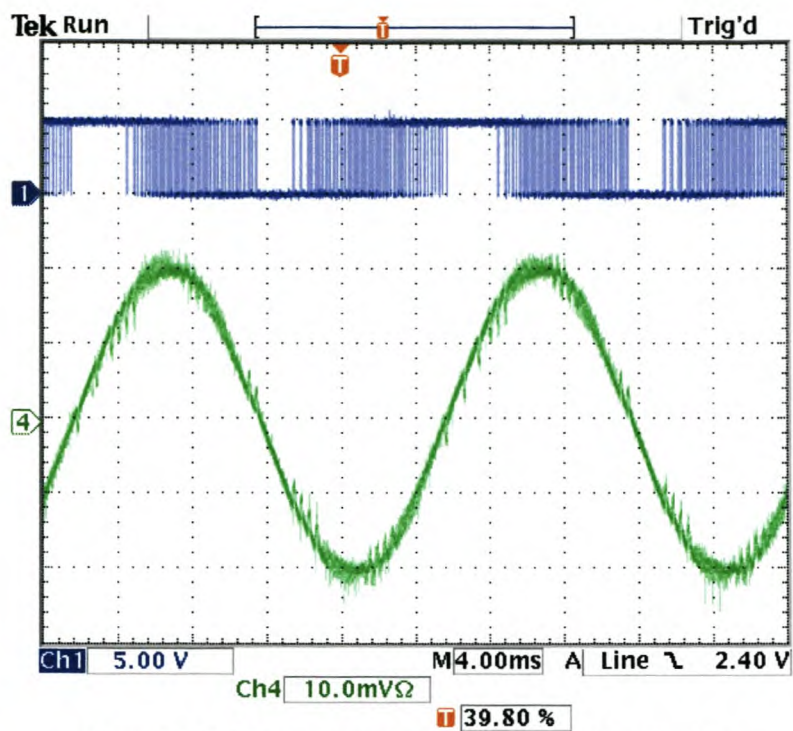


Figure 6-16 – Current regulation using module controller

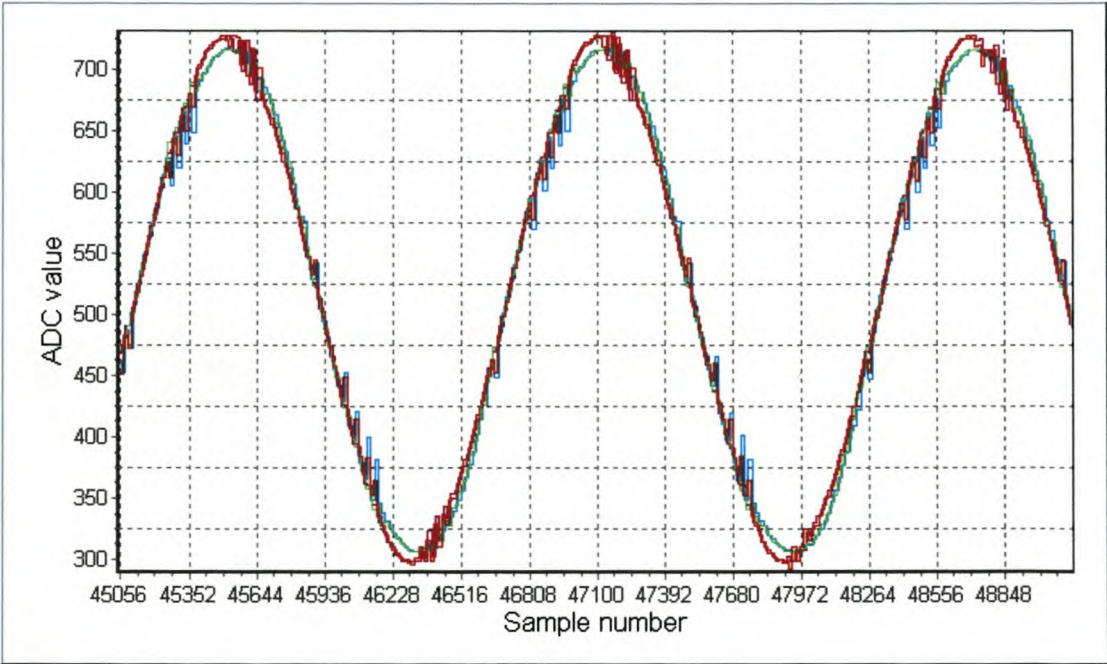


Figure 6-17 – Current measured by module controller (all 4 sampling instants)

6.3.2 Voltage regulator

The module controller is used to regulate the output voltage. The reference voltage is generated internally. The setup for the evaluation is shown in Figure 6-18.



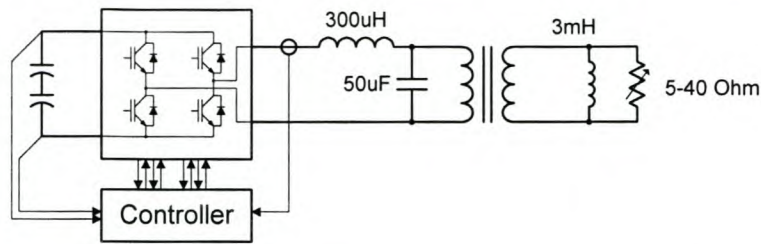


Figure 6-18 – Test setup for voltage regulator configuration

The results are shown in Figure 6-19. Trace two is the voltage and trace four is the current. The current scale is set to 25 A per division. The distortion visible on the voltage waveform is caused by the magnetizing inductance of the transformer and blanking time of the PWM control signals (current becomes discontinuous).

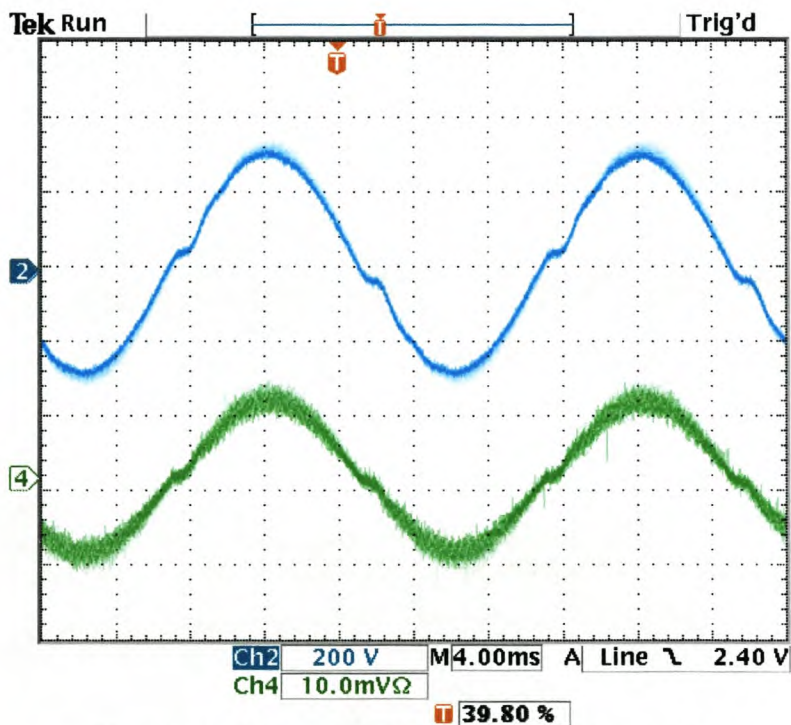


Figure 6-19 – Voltage regulation with module controller

## 6.4 Converter topologies

### 6.4.1 Three-phase with coupled outputs

For this experiment three modules controllers are used. The DC-buses are connected in parallel and only one of the phase-arm outputs of the module is used. The load is connected in star.

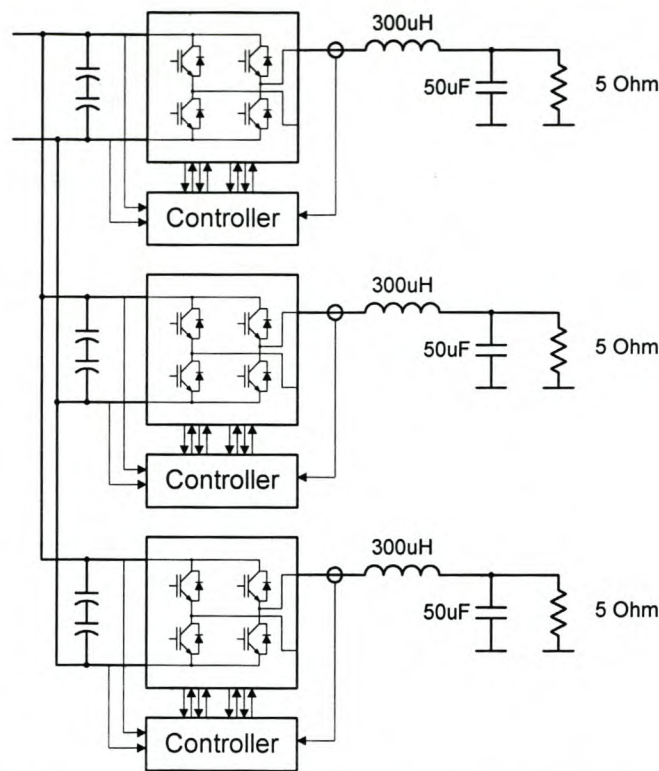


Figure 6-20 – Three-phase with coupled outputs test setup

The results of the three-phase test are shown in Figure 6-21. The phase to neutral voltages of phase B and phase C are shown as trace 1 and trace 2 respectively. Trace 4 shows the phase current of phase A. The DC-bus voltage is 450 V.

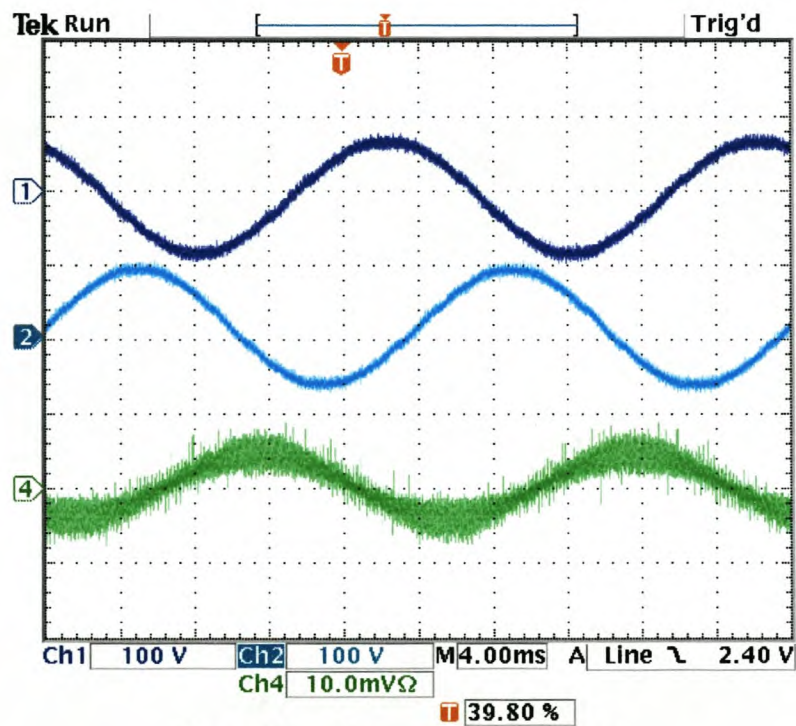
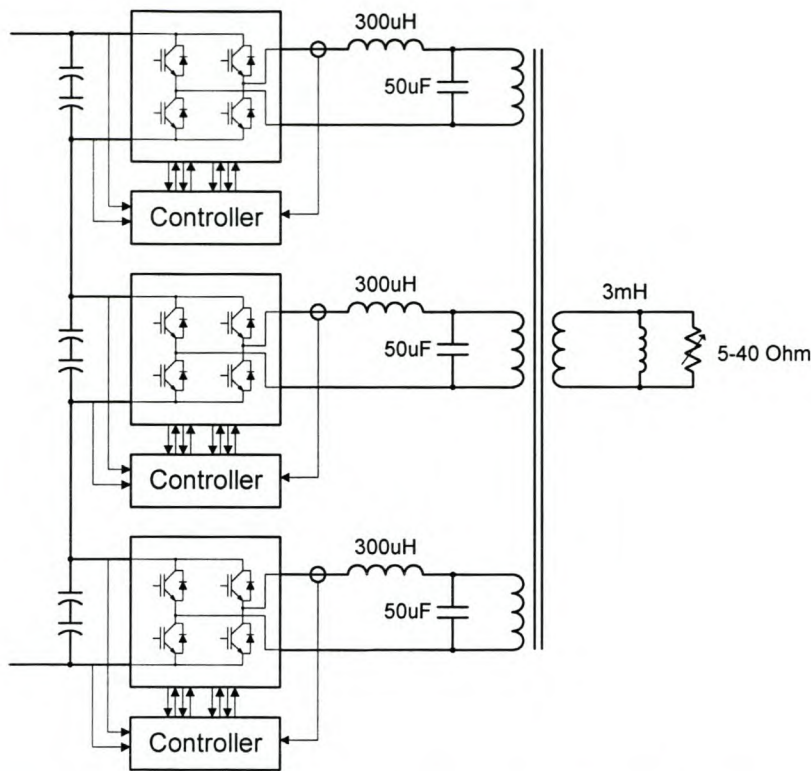


Figure 6-21 – Three-phase converter test



### 6.4.2 Series stacked multilevel



**Figure 6-22 – Series stacked multilevel converter test setup**

The operation of the series stacked converter is shown in Figure 6-23. Traces 1, 2 and 3 show the output currents of the individual cells. Trace 4 shows the combined current at the load. The current scale is 10 A per division.

During open-loop steady-state operation, the three DC-bus voltages should balance [43]. When each cell is individually switched, unbalance can occur due to an uneven distribution of power supplied by the three cells. This can easily be caused by measurement scaling and offset errors. The voltages can be actively balanced by varying the power delivered by the cell. The DC-bus voltage is measured and compared to a reference sent from the master controller. If the bus voltage is too high, the cell must deliver more power to the load. This will cause more energy to be drawn from the DC-bus capacitors, thereby discharging the capacitors and lowering the bus voltage.

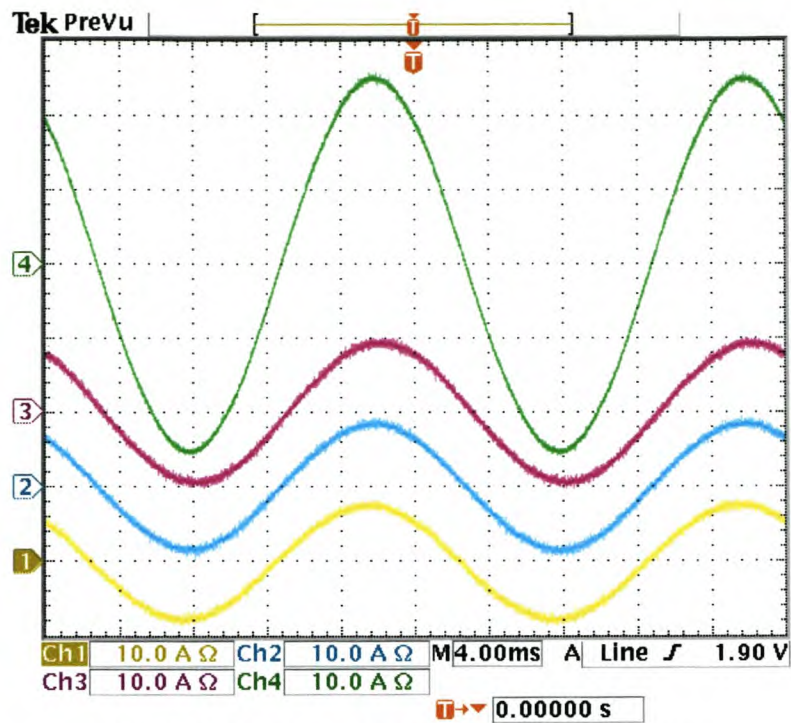


Figure 6-23 – Operation of series stacked converter

The active balancing strategy was tested using the three-level converter shown in Figure 6-22. An additional resistor was used to unbalance the capacitor voltage of one of the cells. The results of natural balancing and the forced balancing are shown in Figure 6-24 and Figure 6-25 respectively. Note the more rapid response of the forced balancing method.

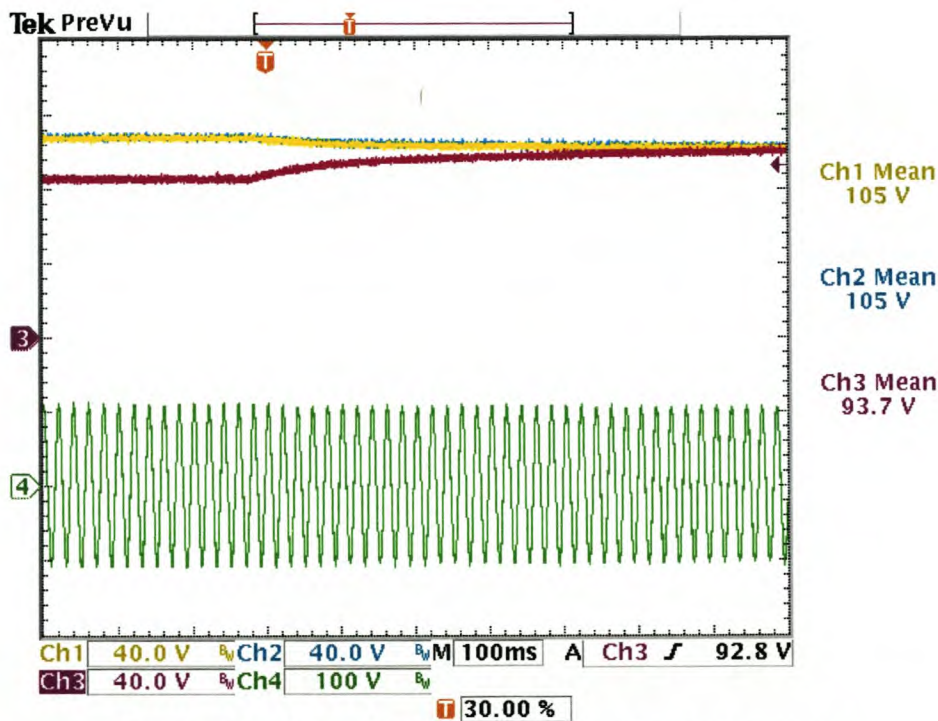


Figure 6-24 – Natural balancing of three-level stacked converter



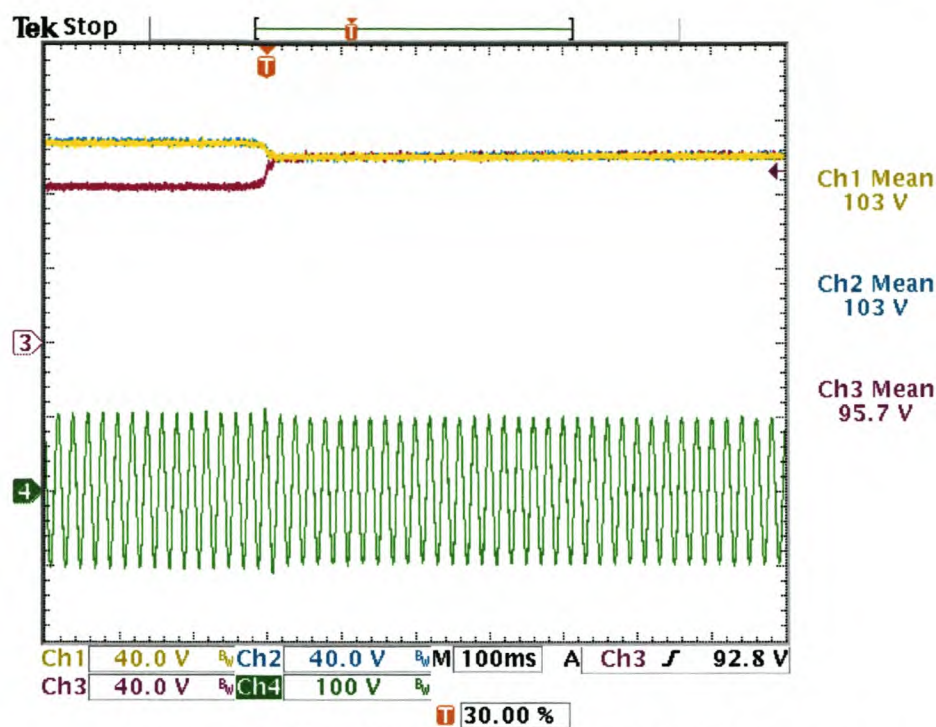


Figure 6-25 – Forced balancing of three-level stacked converter

6.5 Summary

The operation of the module controller was tested in a variety of modes. The individual components were tested to see if they conform to the original specifications stated.

The protection is a major factor when designing the module controller. The distributed control approach requires more stringent protection than the centralized approach. The central controller has the advantage that it has access to all of the status information of the application system. In contrast to this, each distributed module has access to local status information only.

The critical failure of one target module must cause the system to shut down into a safe mode. This was demonstrated in the practical evaluation of the protection.

The three-phase test also shows that the converter can be operated as a standard three-phase converter, with the added advantage of reliable control and status signals and embedded measurements.

# Chapter 7

---

## Conclusions



## Chapter 7 Conclusions

### 7.1 Converter topologies and control

The dissertation gives background on a wide variety of traditional and emerging converter topologies. The internal operation of the converter must be understood before individual cells can be determined. Most of the more complex converters can be constructed from repeatable blocks. The second part of the project was to identify the basic blocks for the construction of multilevel and multicell converters.

Three basic cells were identified, the (1) half-bridge, (2) full-bridge and (3) the imbricated-cell. A study was undertaken to identify the interaction of the cells within a converter.

### 7.2 Distributed control

The basic idea of the power block is to have a self-contained block requiring only power and control inputs. The power block provides all measurements needed for control and protection. The measurements and internal status are made available to an external controller by means of an optic fiber communications link.

The study further identified the requirements for the construction and operation of the generic power block, including the power supply, measurements and communications interface.

An investigation into existing communication protocols for use by the distributed controllers was undertaken and options given for the current implementation.

The generic power block requires a source of power for the controller and gate-drive circuits. Options were given for distributing power to the various modules. The two main choices are AC and DC distribution and the advantages and disadvantages of the two methods were discussed.

The power block is fully self-controlled and can therefore operate in a number of configurations or modes. The modes were identified and the necessary measurement and control specifications were identified.

The final part of the distributed control investigation was to identify the converter topologies that can benefit from distributed control and the implications for stability



and control of the converter as a single unit. Two case studies were given that compared the centralized and distributed implementations.

Work done by Milosavljevic, Celanovic and Boroyevich [32][33][34] also investigated the use of distributed control for power building blocks. The focus of that study leaned more toward implementing a “dumb” slave device, capable of receiving PWM references and generating the appropriate gating signals. This stands in contrast to the intelligent self-controlled unit investigated in this project.

### **7.3 Development of the power block**

The detailed design of a power block was done in Chapter 5. The block is called the module controller and can perform all of the functions specified in the preceding chapters.

Special care had to be taken with the protection of the module controllers. A centralized controller has access to all of the data of the system and can take the appropriate action to bring the system to a safe operating state. For the distributed implementation, each module has access to the local status and measurements only. A fault condition in a block lower down the communications link can cause a catastrophic failure if all of the modules are not brought to a safe state. The protection was implemented by using special idle detection on the high-speed fiber link. The response time was measured at less than 2  $\mu$ s. This should be sufficient for even the more demanding systems.

To enable debugging of the system, a data acquisition sub-system and the online download of data were implemented within the embedded software. A client/server application was written to communicate with the controllers via a local personal computer or local area network (LAN). The code was written to be fully layered and generic, allowing the client and server to use different signalling paths and target different types of devices.

### **7.4 Evaluation**

The operation of the module controller was tested in a variety of modes and conditions. The controllers operated as expected and within specifications.



The low-level synchronization is an important factor for optimal control and stability of certain of the converter topologies. The high-speed fiber link was also used to enable the module controllers to stay within 200 ns of the master controller timing.

## 7.5 Comments and future work

*Controller hardware:* The hardware developed for the controller can be simplified significantly. Newer versions of the digital signal processor used for control incorporate sufficient memory for the data acquisition unit. The programmable logic of the module controller must be replaced with a low-cost field-programmable gate array to allow a more complete implementation of the required communications link.

*Communications link:* Alternatives for the custom fiber link are also becoming available, for example, the newer MACRO and SERCOS links. These links provide a standard that is supported by industry.

*Cascade multilevel converter:* The cascade multilevel converter seems to be a viable solution for medium-power applications. A more detailed analysis of control options might lead to a viable distributed control strategy for the cascade converter.

One option might be to assign a stage number to each consecutive cell. The cell then uses the number to determine the reference level it should respond to. After each half-cycle of the fundamental, the stage numbers are increased (modulo- $n$  where  $n$  is the number of cells). During the next half-cycle, the next cell in the chain will respond to the reference level of the previous cell. This should allow for sharing of the energy drawn from the DC supplies. The module controller can then handle the sharing of the conduction losses of the switches of the full-bridge converter. This strategy is similar to that proposed by [30] for sharing of the energy storage utilization.

# Chapter 8

---

## References



## Chapter 8 References

### Textbooks

- [1] N. Mohan, T.M. Undeland, and W.P. Robbins, *Power Electronics: Converters, Applications, and Design*, 2nd edition New York: Wiley, 1995.
- [2] P.C. Sen, *Principles of Electric Machines and Power Electronics*, New York: John Wiley and Sons, 1989.
- [3] F. Halsall, "Data Communications, Computer Networks and Open Systems," Addison-Wesley, 1992.
- [4] A.L. Preston, "Switching and Linear Power Supply, Power Converter Design," Hayden Book Company, 1997.
- [5] M.I. Montrose, "Printed Circuit Board Design Techniques for EMC Compliance," IEEE Press, 1996.

### Journal papers

- [6] D.M. Divan, "The resonant DC link converter – A new concept in static power conversion," *IEEE Transactions on Industry Applications*, vol. 25, pp. 317-325, March/April 1989.
- [7] G.J. Joos, X. Huang, and B.T. Ooi, "Direct-coupled multilevel cascaded series VAR compensators," *IEEE Transactions on Industry Applications*, vol. 34, no. 5, pp. 1156-1163, Sept./Oct. 1998.
- [8] J. Kuang, and B.T. Ooi, "Series connected voltage-source converter modules for force-commutated SVC and DC-transmission," *IEEE Transactions on Power Delivery*, vol. 9, no. 2, pp.977-983, April 1994.
- [9] W. McMurray, "Optimum snubbers for power semiconductors," *IEEE Transactions on Industry Applications*, vol. IA-8, no. 5, pp. 593-600, Sept./Oct. 1972.
- [10] T.A. Meynard, M. Fadel, and N. Aouda, "Modelling of multilevel converters," *IEEE Transactions on Industrial Electronics*, vol. 44, no. 3, pp. 356-364, June 1997.

- [11] H. duT. Mouton, and J.H.R. Enslin, "An optimal on-line-tuning current regulator for high-power IGBT converters," IEEE Transactions on Industry Applications, vol. 35, no. 5, pp. 1132-1140, Sept-Oct. 1999.
- [12] A. Nabae, I. Takahashi, and H. Akagi, "A new neutral-point-clamped PWM inverter," IEEE Transactions on Industry Applications, vol. IA-17, no. 5, pp. 518-523, Sept./Oct. 1981.
- [13] B. Mwinyiwiwa, B.T. Ooi, and Z. Wolanski, "UPFC using multi-converters operated by phase-shifted triangle carrier SPWM strategy," IEEE Transactions on Industry Applications, vol. 34, no. 3, pp. 495-500, May-June 1998.
- [14] B. Mwinyiwiwa, Z. Wolanski, and B.T. Ooi, "Microprocessor-implemented SPWM for multi-converters with phase-shifted triangle carriers," IEEE Transactions on Industry Applications, vol. 34, no.3, pp. 487-494, May-June 1998.
- [15] G. Venkataramanan, and D.M. Divan, "Pulse width modulation with resonant DC-link converters," IEEE Transactions on Industry Applications, vol. 29, pp. 113-120, Jan./Feb. 1993.
- [16] Z. Zhang, J. Kuang, X. Wang, and B.T. Ooi, "Force commutated HVDC and SVC based on phase-shifted multi-converter modules," IEEE Transactions on Power Delivery, vol.8, no.2, pp.712-718, April 1993.
- [17] H. Akagi, A. Nabae and S. Atoh, "Control strategy of active power filters using multiple voltage-source PWM converters," IEEE Transactions on Industry Applications, vol. 22, no. 3, May-June 1986, pp. 460-465.
- [18] V.A.K. Temple, "MOS-controlled thyristors, a new class of power devices," IEEE Transactions on Electronic Devices, vol. ED-33, no. 10, Oct. 1986, pp. 1609-1618.
- [19] D.M. Divan, "The resonant DC link converter, a new concept in static power conversion," IEEE Transactions on Industry Applications, vol.25, no.2, March-April 1989, pp.317-325.
- [20] B.K. Bose, "Evaluation of modern power semiconductor devices and future trends of converters," IEEE Transactions on Industry Applications, vol.28, no.2, March-April 1992, pp.403-413.



### Conference papers

- [21] D.D. Bester, J.A. du Toit, and J.H.R. Enslin, "A DSP controller for power electronic converters," IEEE International Symposium on Industrial Electronics (ISIE'98), Pretoria, South Africa, July 7-10, 1998, pp. 265-279.
- [22] H.J. Beukes, J.H.R. Enslin, and R. Spee, "Active snubber for high power IGBT modules," in IEEE AFRICON Conference Recordings, Sept. 1996, vol. 1, pp. 456-461.
- [23] R.W. De Doncker, J.P. Lyons, "The auxiliary resonant commutated pole converter," Conference Proceedings IEEE-IAS Annual Meeting, 1990, pp. 1128-1235.
- [24] R.W. De Doncker, J.P. Lyons, "The auxiliary quasi-resonant DC link inverter," IEEE Power Electronics Specialist Conference (PESC-1991), June 1991, pp. 248-253.
- [25] J.A. du Toit, D.D. Bester, J.H.R. Enslin, "A DSP based controller for back-to-back power electronic converters with FPGA implementation," IEEE Applied Power Electronics Conference (APEC-97), 1997, pp. 699-705.
- [26] H. duT. Mouton, J.H.R. Enslin, "A high power IGBT based series injection power quality conditioner," IEEE International Conference on Harmonics and Quality of Power (ICHQP-98), Athens, Greece, Oct. 14-16, 1998, pp. 203-209.
- [27] A.J. Visser, H. duT. Mouton, "250 kW Transformer-less voltage dip compensator," in Conference Rec. IEEE AFRICON-1999, Cape Town, South Africa, Sept., 1999, vol. 2, pp. 865-880.
- [28] A.J. Visser, H duT. Mouton, J.H.R. Enslin, "Direct-coupled multilevel sag compensator," IEEE Power Electronics Specialist Conference (PESC-2000).
- [29] L.M. Tolbert, F.Z. Peng, T.G. Habetler, "Multilevel PWM methods at low modulation indices," IEEE Applied Power Electronics Conference (APEC-1999), 1999, pp. 1032-1039.
- [30] L.M. Tolbert, F.Z. Peng, "Multilevel converters for large electric drives," IEEE Applied Power Electronics Conference (APEC-1998), pp. 530-536.



- [31] B-S. Suh, Y-H. Lee, D-S. Hyun, T.A. Lipo, "A new multilevel inverter topology with a hybrid approach," (EPE-1999), 1999.
- [32] I. Celanovic, N. Celanovic, I. Milosavljevic et al, "A New Control Architecture for Distributed Power Electronics Systems," IEEE Power Electronics Specialist Conference (PESC-2000).
- [33] I. Celanovic, I. Milosavljevic, D. Boroyevich et al, "A New Distributed Controller for Next Generation Power Electronics Building Blocks," IEEE Applied Power Electronics Conference (APEC-2000), Feb 2000, pp. 889-894.
- [34] I. Milosavljevic, Z. Ye, D. Boroyevich, C. Holton, "Analysis of Converter Operation with Phase-Leg in Daisy-Chained or Ring-Type Structure," IEEE applied Power Electronics Conference (APEC-1999), pp. 1216-1221.

### **Data books and application notes**

- [35] POWEREX, "IGBTMOD and Intellimod TM - Intelligent Power Modules," Applications and Technical Data Book, Nov. 1994.
- [36] Philips, "Soft Ferrites," Data Handbook MA01, 1996.
- [37] Semikron, "Power Electronics," Data Book, 1997.
- [38] Hewlett Packard, "Optoelectronics Designer's Catalogue," 1993.
- [39] Agilent Technologies application note 1121, "Inexpensive DC to 32 MBd fiber-optic solutions for industrial, medical, telecom and proprietary data communication applications."
- [40] Agilent Technologies application note 1122, "Inexpensive 2 to 70 MBd fiber-optic solutions for industrial, medical, telecom and proprietary communication applications."
- [41] Texas Instruments "Optical Networking Solutions Guide", 4<sup>th</sup> quarter 2001.

### **Dissertations**

- [42] H.J. Beukes, "Synthesis of a high-performance power converter for electric distribution applications," PhD Thesis, University of Stellenbosch, 1997.
- [43] H duT. Mouton, "Analysis and Synthesis of a 2 MVA Series-stacked Power-Quality Conditioner," PhD Thesis, University of Stellenbosch, 2000



- [44] C. Putter, "Development of a high-voltage converter for a DC machine drive application," Masters Thesis, University of Stellenbosch, 1997.

### **Web sites**

- [45] Altera: <http://www.altera.com/>
- [46] Analog Devices: <http://www.analog.com/>
- [47] LEM: <http://www.lem.com/>
- [48] MACRO: <http://www.macro.org/>
- [49] SERCOS: <http://www.sercos.com/>
- [50] IEEE1394: trade association <http://www.1394ta.org/>
- [51] CAN trade association: <http://www.can-cia.de/>
- [52] USB developers: <http://www.usb.org/>
- [53] Infrared Data Association: <http://www.irda.org/>
- [54] Texas Instruments: <http://www.ti.com/>

# Appendix A

---

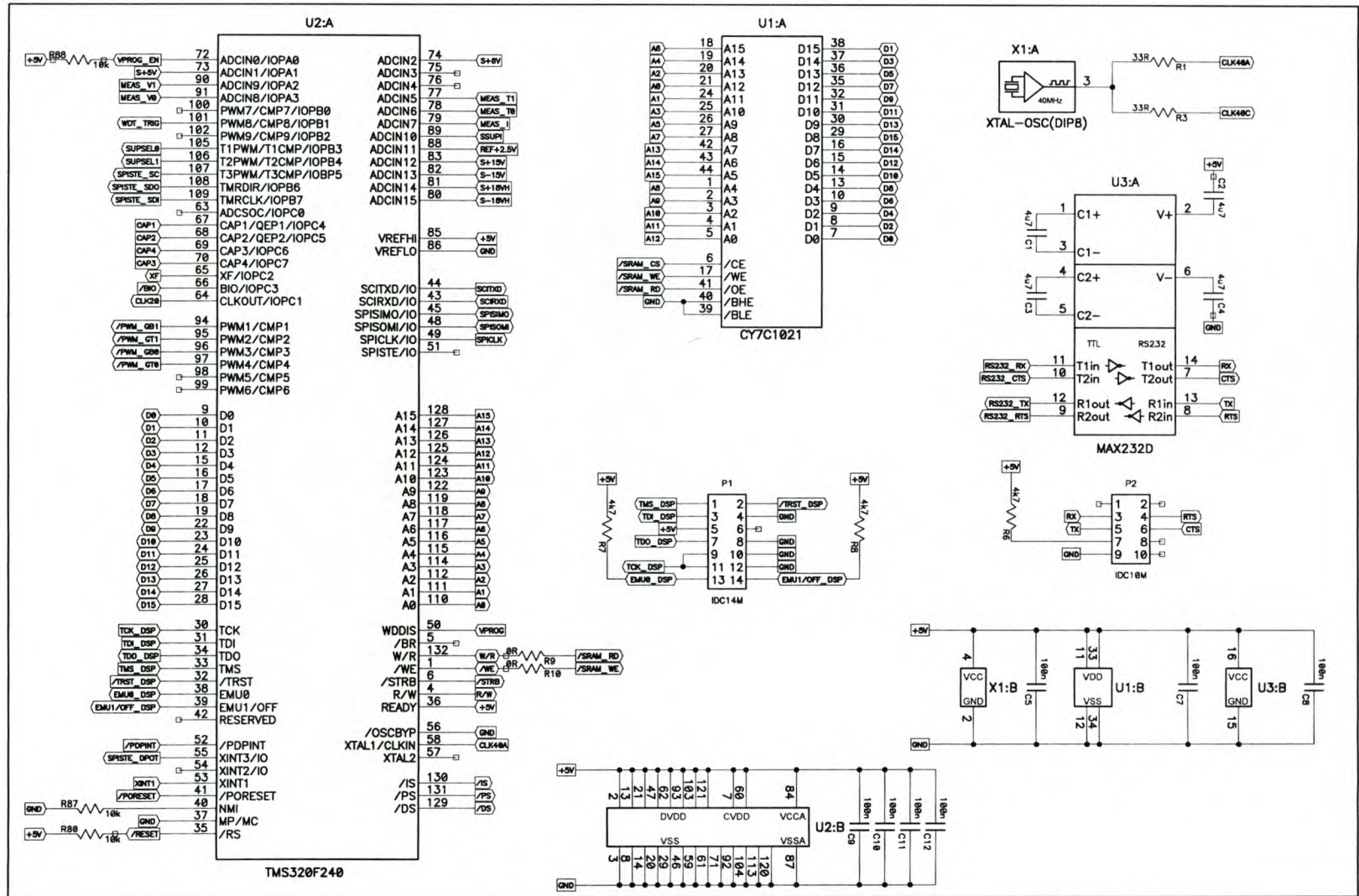
## Modcon schematics



## A. Modcon schematics

The schematics of the following boards are included in this appendix:

- (1) Module controller main board with measurement add-on
- (2) Distributed supply main bridge
- (3) Gate interface protection and transformers





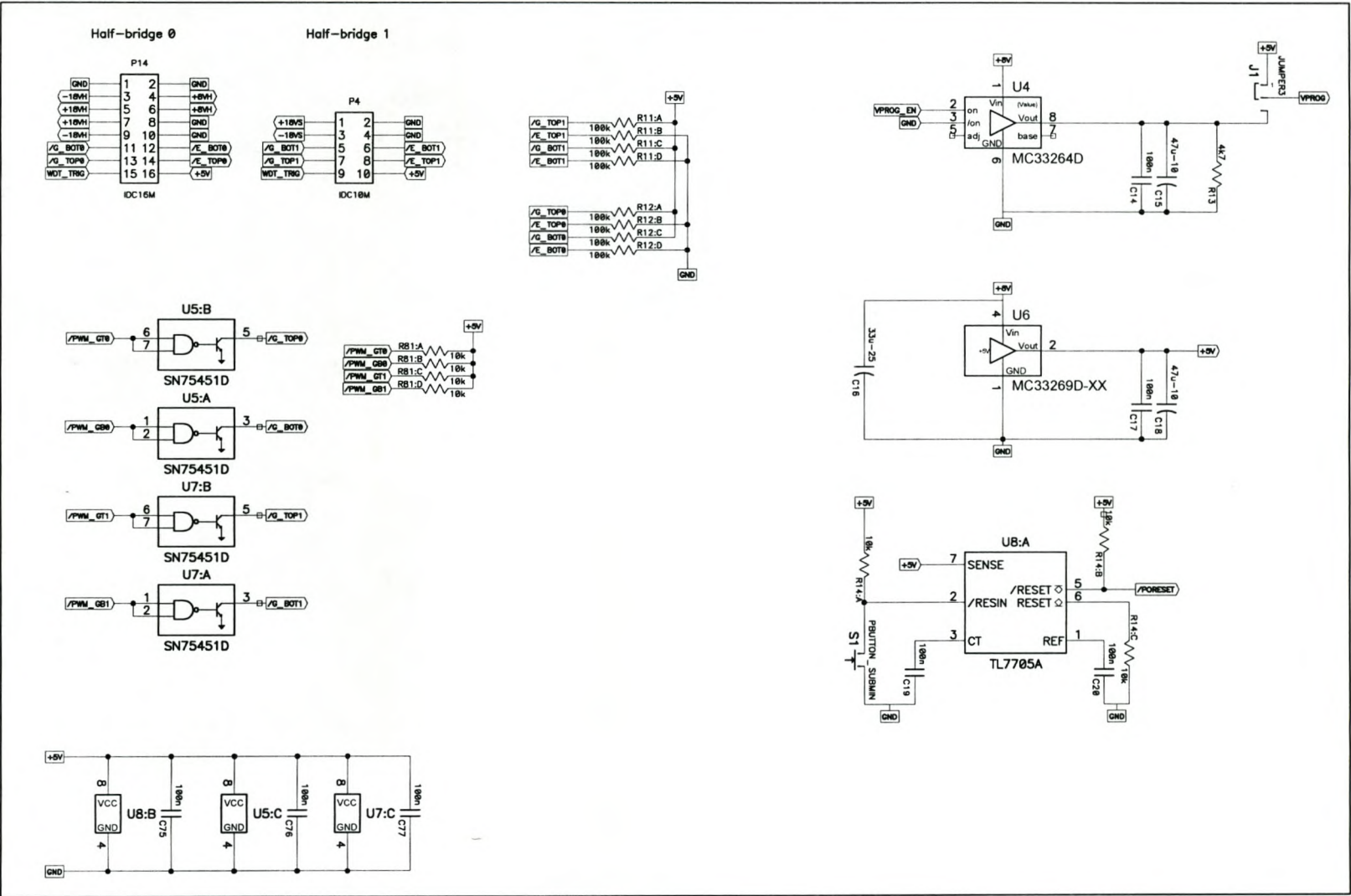
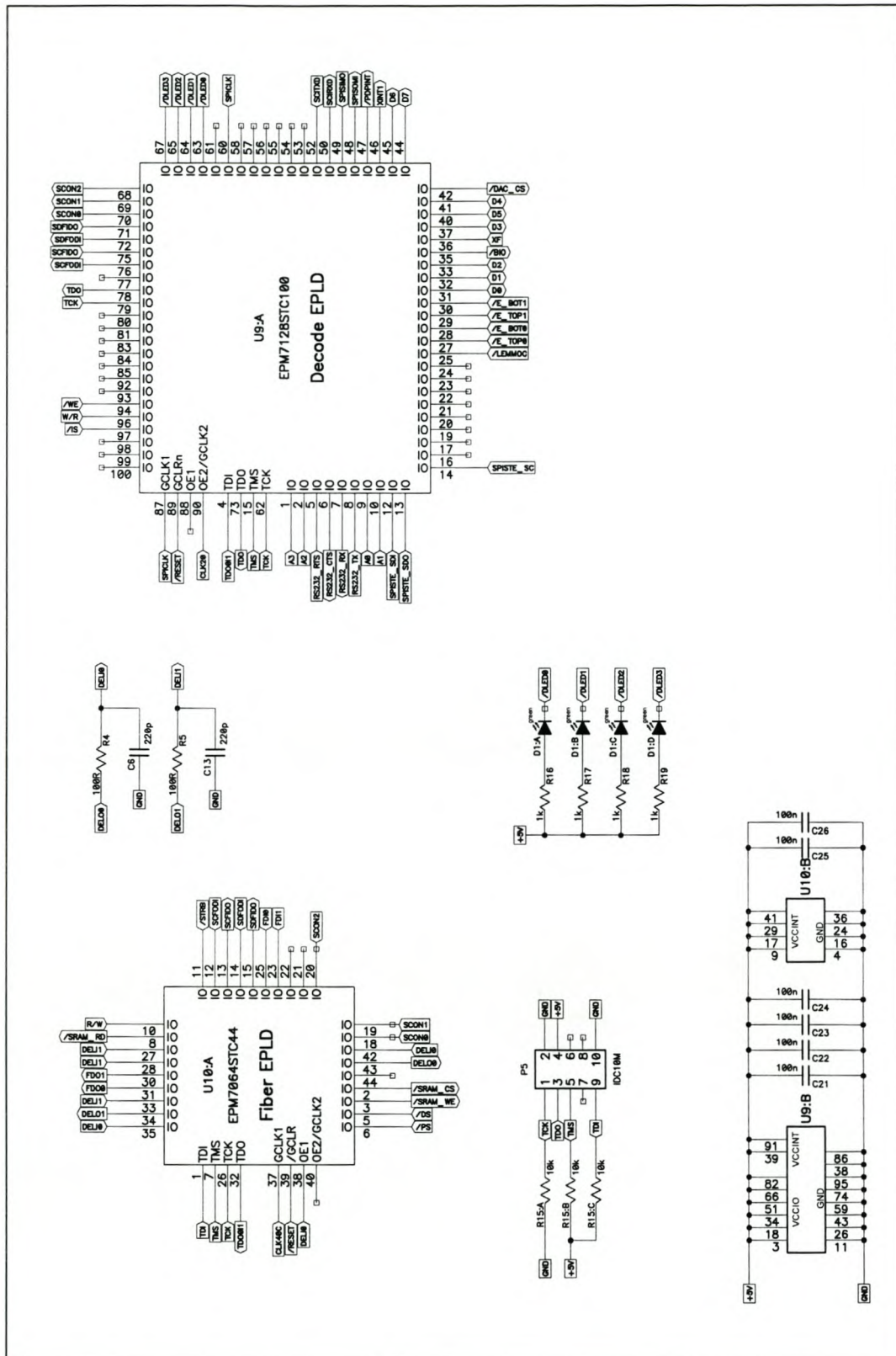


Figure 8-2 – Module controller (Power)





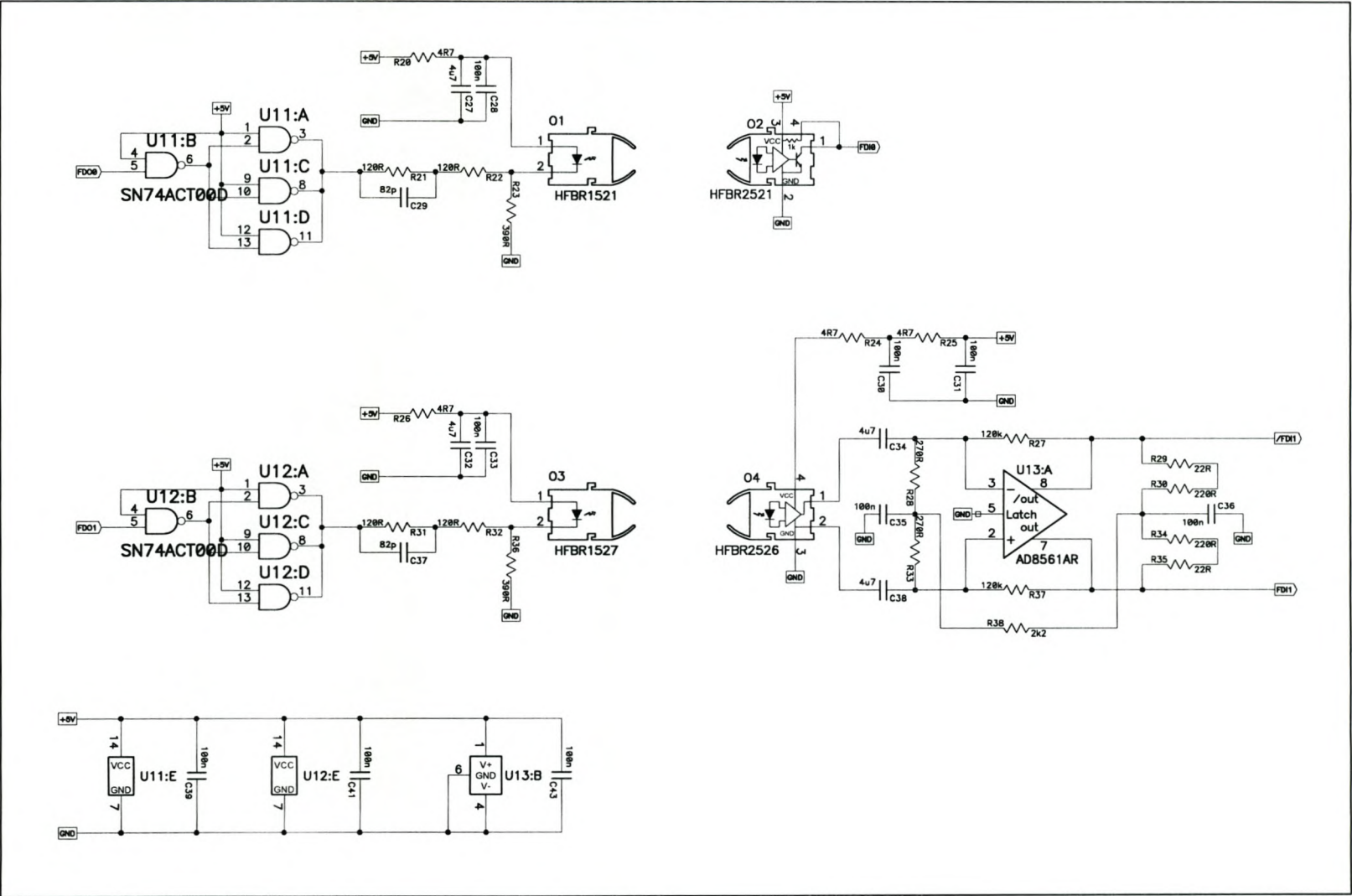


Figure 8-4 – Module controller (Fiber optic)

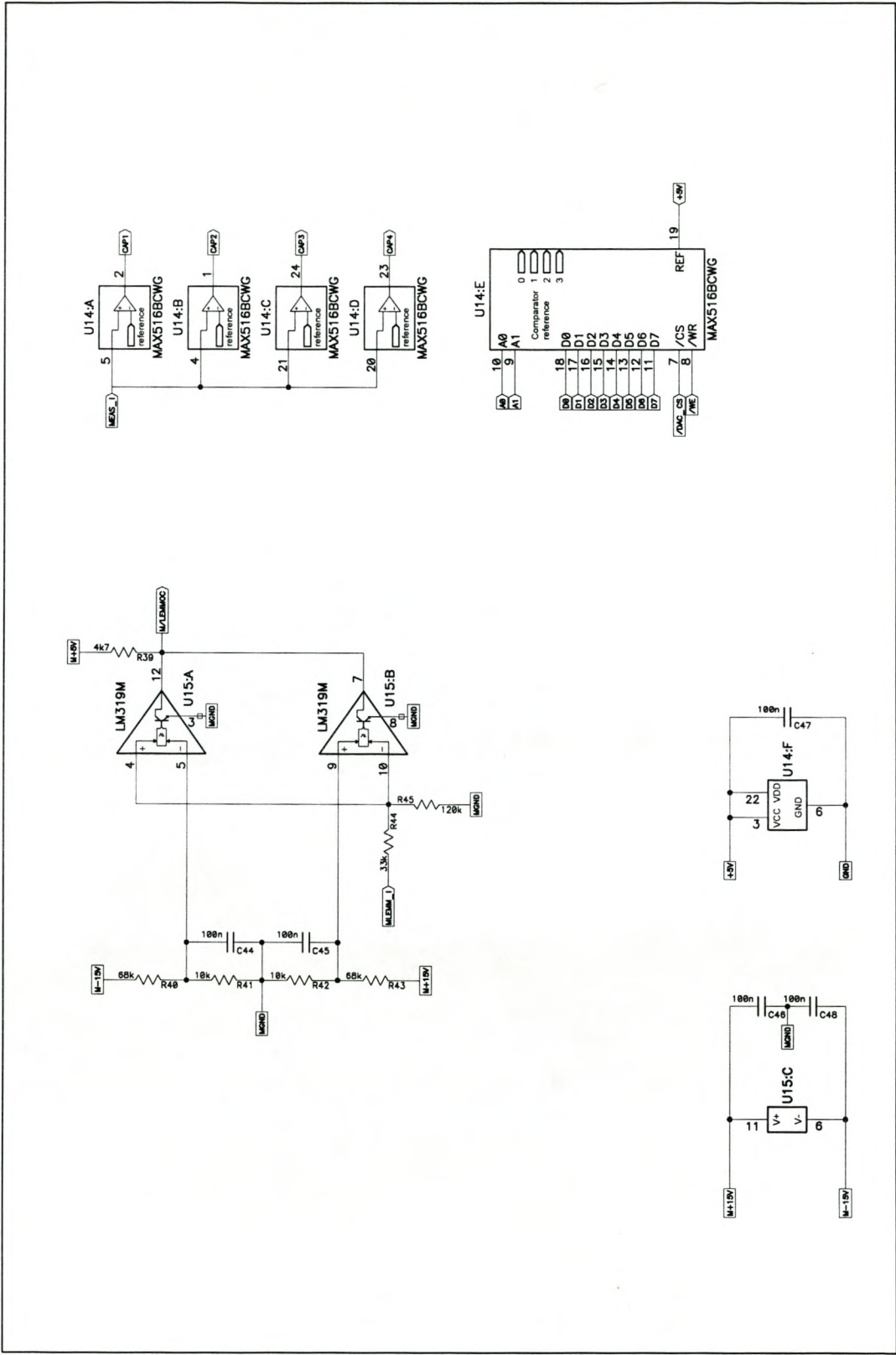
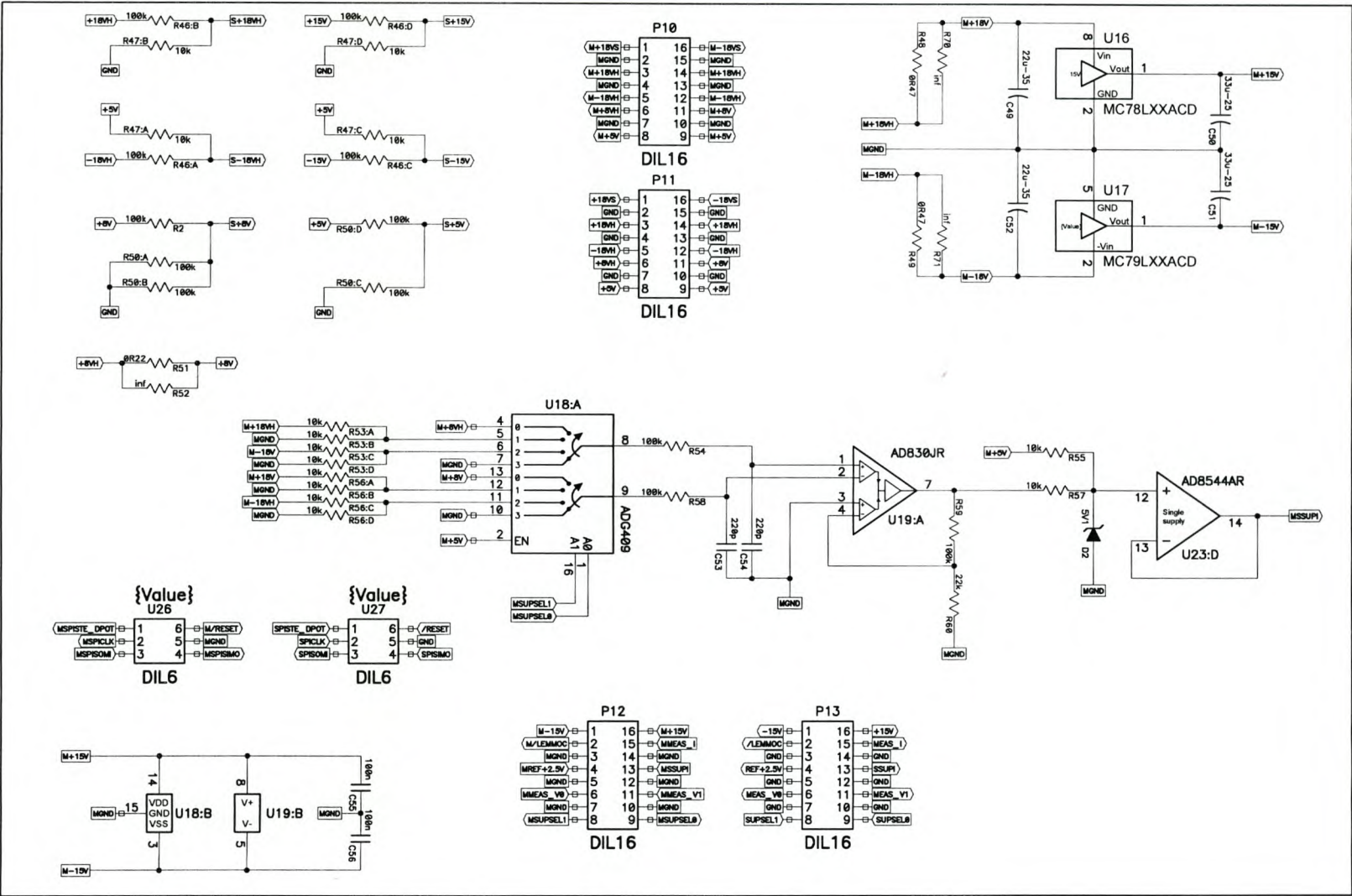


Figure 8-5 – Module controller (Protection)



Figure 8-6 – Module controller (Supply measurements)



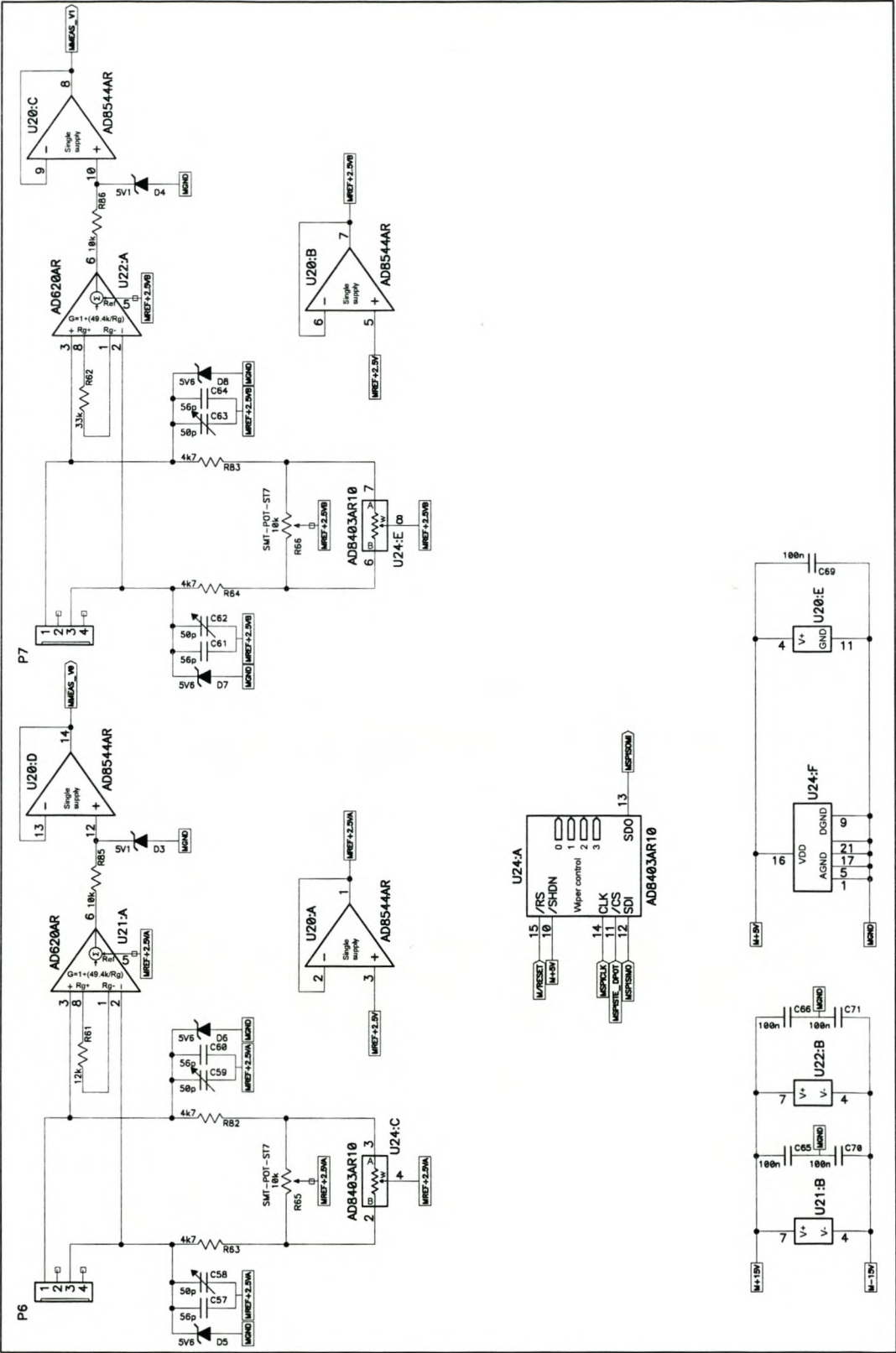


Figure 8-7 – Module controller (Voltage measurements)



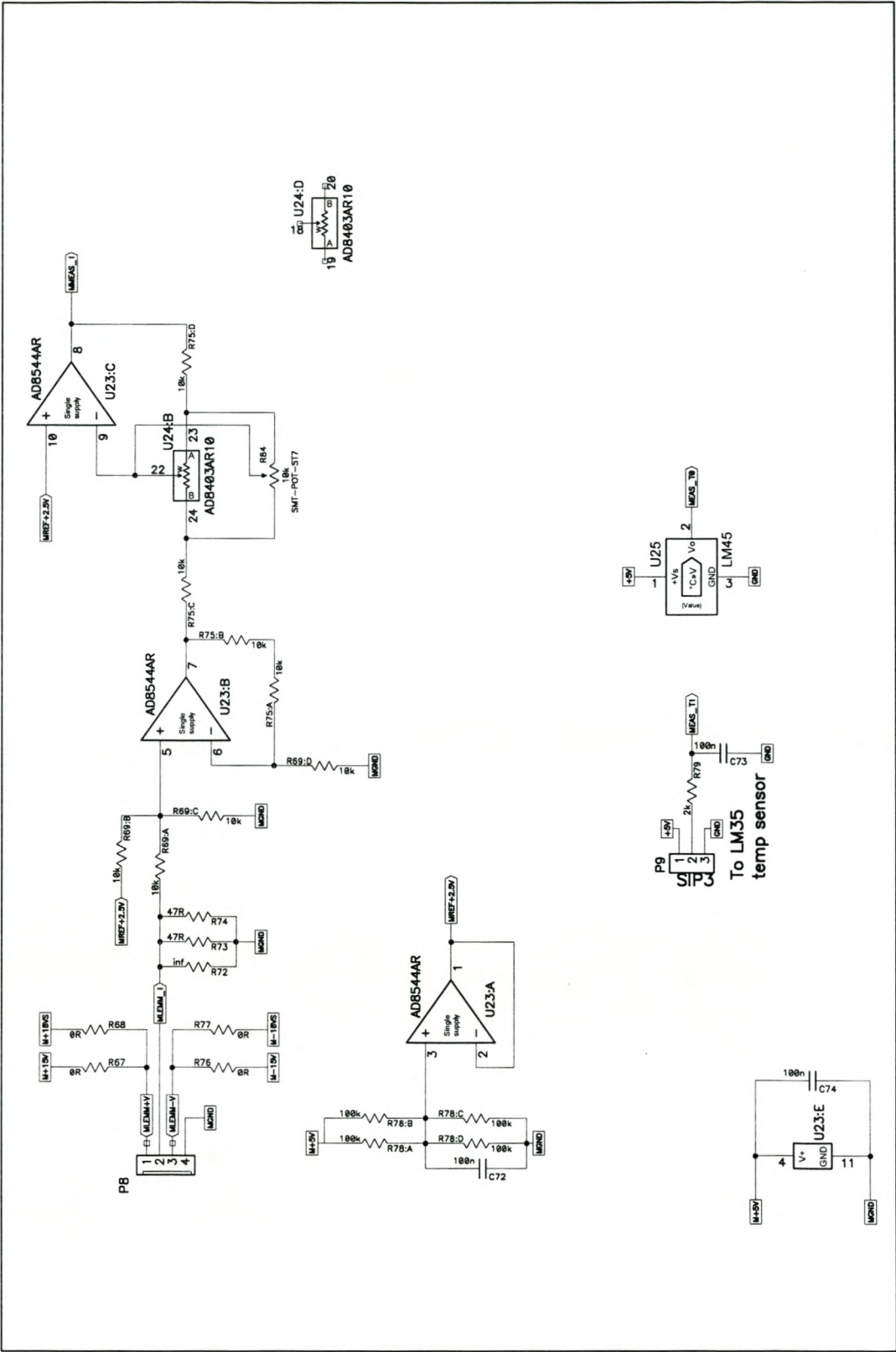
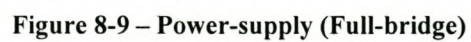
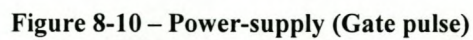


Figure 8-8 – Module controller (Current measurement)







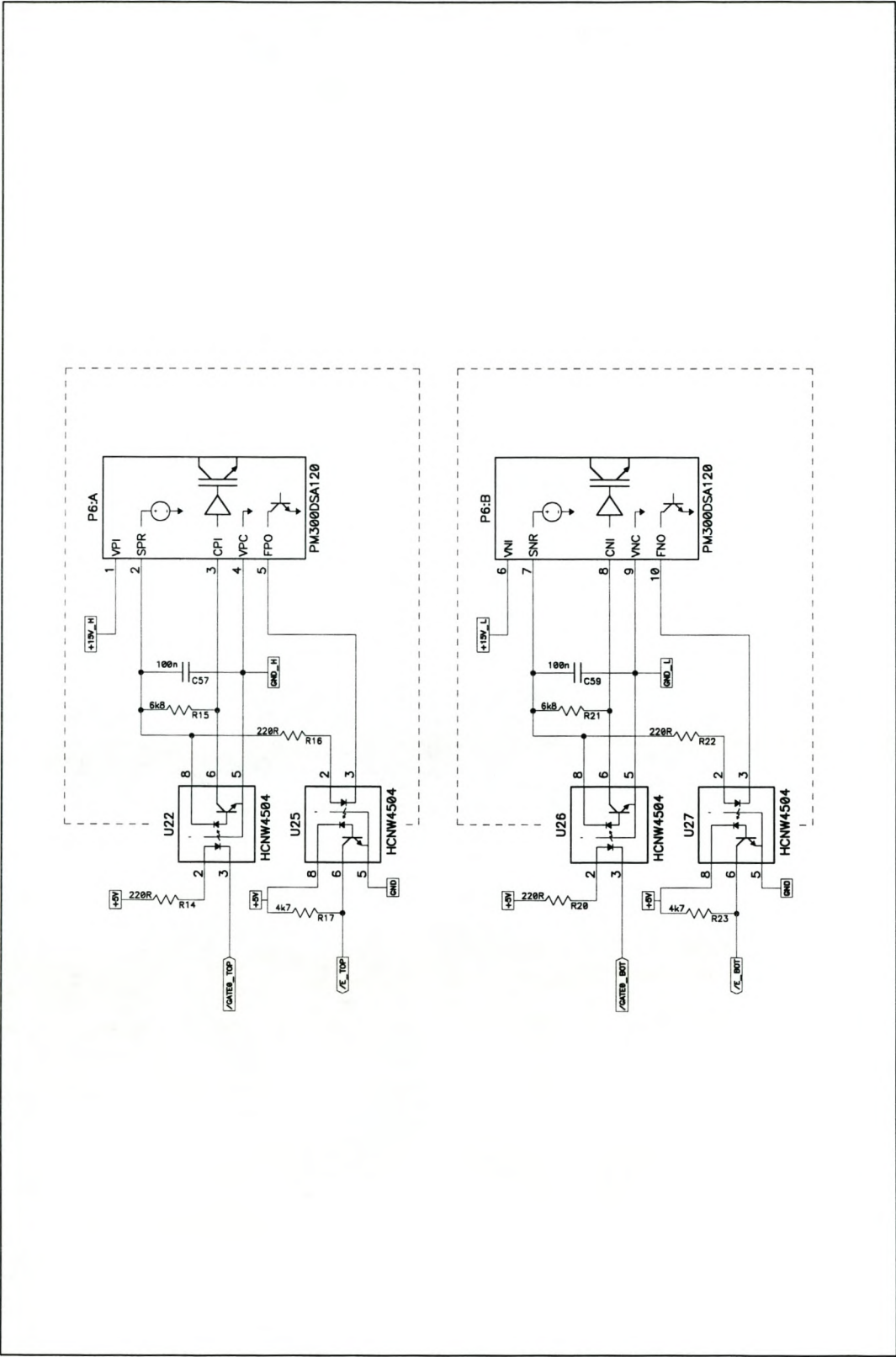


Figure 8-11 – Gate interface (Isolation)



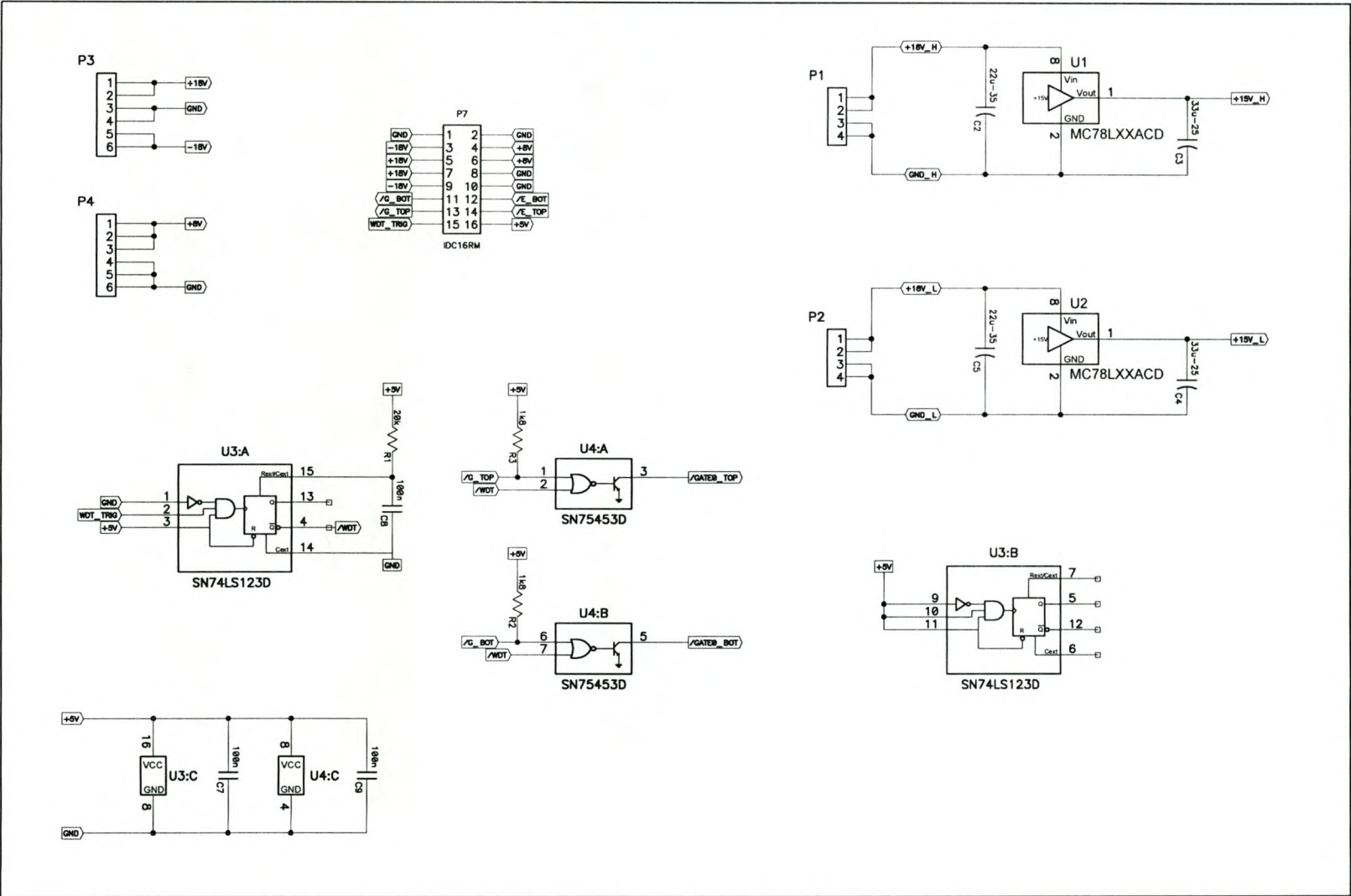


Figure 8-12 – Gate interface (Power)

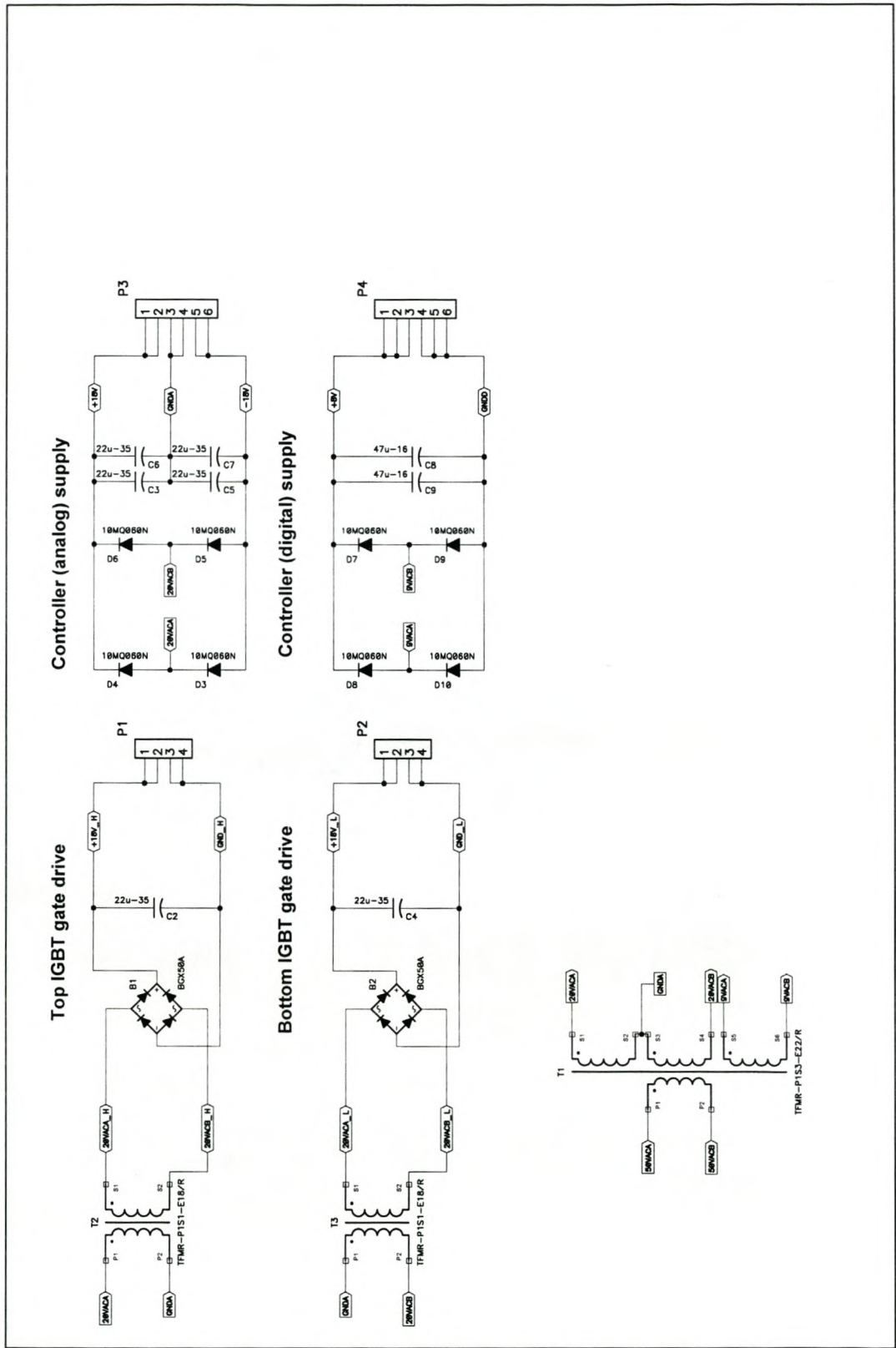


Figure 8-13 – Gate interface (Transformers)



# Appendix B

---

---

PLD firmware source

## B. PLD firmware source

This appendix lists the source code for the EPLDs on the modcon main controller. The EPLDs are called *fiber* and *decode* and handles the fiber optic interface and external decoding respectively.

The EPLDs for the master and slave controllers differ in the functioning of the synchronization blocks. The master is responsible for generating the pulse trains required for the reference sync, carrier sync and line break conditions.

### ModCon master fiber EPLD main source file

```
%
    ---< ModCon Master fiber epld >---
    -----
    file:      mfiber.tdf
    project:   Module Controller (ModCon)
    developer: J.A. du Toit
    company:   University of Stellenbosch
    -----

    2001/08/21 (ver.1.0.0)
    * generate separate carrier and reference sync pulses

    2001/08/15 (ver.0.2.0)
    * Use fdil, fdol as sync pulse train

    2001/05/28 (ver.0.1.0):
    * Changed SFLMC10 rx data receive length to 8 bits
    -----

%
INCLUDE "sflmc10a_dec.inc";

INCLUDE "lpm_counter.inc";
INCLUDE "edge_detect.inc";

INCLUDE "pulse_et.inc";
INCLUDE "pulse_lt.inc";

DEFINE MAX(a,b) = (a > b) ? a : b;
DEFINE WIDTH(a) = CEIL(LOG2(a));

CONSTANT SYNC_GEN_LENGTH = 16;
CONSTANT SYNC_SIG_LENGTH = 7;
CONSTANT SYNC_DET_LENGTH = 10;

CONSTANT BREAK_DET_LENGTH = 31;
CONSTANT BREAK_SIG_LENGTH = 15;
-----

SUBDESIGN mfiber
(
    r/w          : INPUT = VCC;      -- DSP read/write
    /strb        : INPUT = VCC;      -- DSP memory/IO strobe
    clk40        : INPUT = GND;      -- Main clock output
    /ds          : INPUT = VCC;      -- data space
    /ps          : INPUT = VCC;      -- program space

    /reset       : INPUT = VCC;      -- board reset from/to DSP

    /sram_cs     : OUTPUT;           -- sram chip select
    /sram_we     : OUTPUT;           -- sram write enable
    /sram_rd     : OUTPUT;           -- sram output enable

    -- fiber optics
    fdi[1..0]    : INPUT = GND;      -- input from optic fiber rx
    fdo[1..0]    : OUTPUT;           -- output to optic fiber tx

    -- SFL
    tx_clk       : INPUT = GND;      -- serial receive clk (use as ref sync gen)
    tx_sd        : INPUT = GND;      -- serial transmit data (use as sync ena)
    rx_clk       : OUTPUT;           -- serial transmit clk (use as carrier sync detect)
    rx_sd        : OUTPUT;           -- serial receive data (use as ref sync detect)
    sCon        : OUTPUT;           -- serial control out (use as line break detect)

    -- delay signals
    deli[1..0][2..0] : INPUT = GND;  -- delay signals input
)
```



```

delo[1..0]          : OUTPUT;          -- delay signals output

-- SCI fiber
tx_sci              : INPUT = GND;      -- sci fiber transmit data
rx_sci              : OUTPUT;          -- sci fiber received data

-- reserve unused pins
res_[2..0]          : INPUT = GND;

master_sync         : INPUT = VCC;      -- sync from DSP
)

-----
VARIABLE

link_break_det      : pulse_et WITH (LENGTH = BREAK_DET_LENGTH,
                                   TRIGGER_DEPTH = 1,
                                   TRIGGER_EDGE = "both" );

link_break_sig      : pulse_lt WITH (LENGTH = BREAK_SIG_LENGTH);

sync_pulse_gen      : pulse_et WITH (LENGTH = SYNC_GEN_LENGTH,
                                   TRIGGER_DEPTH = 1,
                                   TRIGGER_EDGE = "both" );

sync_osc_counter    : lpm_counter WITH (LPM_WIDTH = 3);

sync_osc            : NODE;
sync_in             : NODE;
sync_out            : NODE;

master_sync_level   : NODE;

ref_sync_ena        : NODE;
sync_ena            : NODE;
carrier_sync_det    : NODE;
ref_sync_det        : NODE;
line_break_det      : NODE;

BEGIN

-----< misc >-----

-- external SRAM (decode in both program AND data space)
/sram_we = r/w # /strb;
/sram_rd = !r/w # /strb;
/sram_cs = /ds & /ps;

-----< SCI >-----

fdo0 = tx_sci;      -- fiber optic out = SCI transmit
rx_sci = !fdi0;     -- SCI receive (inverted if first device in chain)

-----< SFL >-----

-- delay signals
delo0 = GND;
delo1 = GND;

-- connect I/O pins
ref_sync_ena = DFFE(tx_clk, clk40,,, sync_pulse_gen.trig_out);
sync_ena = tx_sd;
rx_clk = carrier_sync_det;
rx_sd = ref_sync_det;
scon = link_break_sig.q;

fdol = DFF(sync_out & sync_ena, clk40,,);
sync_in = DFF(fdol, clk40,,);

carrier_sync_det = GND;
ref_sync_det = GND;

master_sync_level = DFFE(master_sync, clk40,,, sync_pulse_gen.trig_out);

-- sync pulse generator (triggered from DSP)
sync_pulse_gen.clk = clk40;
sync_pulse_gen.d = master_sync;
sync_pulse_gen./reset = /reset;

-- sync pulse generation
CASE (sync_pulse_gen.q, master_sync_level, ref_sync_ena) IS
  WHEN B"0xx" =>
    sync_out = sync_osc;
  WHEN B"100" =>
    sync_out = sync_osc;
  WHEN B"101" =>
    sync_out = GND;          -- sync low
  WHEN B"11x" =>
    sync_out = VCC;         -- sync high
  WHEN OTHERS =>
    sync_out = GND;
END CASE;

-- link break detect and signal pulse generators
link_break_det.clk = clk40;
link_break_det.d = sync_in;
link_break_det./reset = /reset;

line_break_det = !link_break_det.q;

```

```

link_break_sig.clk = clk40;
link_break_sig.d = !link_break_det.q;
link_break_sig./reset = /reset;

-- sync oscillator
sync_osc_counter.clock = clk40;
sync_osc_counter.sset = sync_pulse_gen.trig_out;
sync_osc = DFF(sync_osc_counter.q2, clk40,,);

```

```

END;

```

## **ModCon master decode EPLD main source file**

```

%          ----< ModCon Master decode EPLD >----
%
%
file:      mdecode.tdf
project:   Module Controller (ModCon)
developer: J.A. du Toit
company:   University of Stellenbosch
%
%
CONSTANT   epld_version      = 040;
%
%
2001/08/21 (ver.0.4.0):
· generate separate carrier and reference sync pulses
· added enable bits to control register to select pdpint sources

2001/08/20 (ver.0.3.0):
· added control and status registers for protection
· reference sync out moved to xint

2001/08/15 (ver.0.2.0):
· Use fdil, fdol as sync pulse train

2001/06/01 (ver.0.1.0):
· Initial version
%
%
include "lpm_counter.inc";
include "lpm_shiftreg.inc";

include "edge_detect.inc";
INCLUDE "pulse_et.inc";

DEFINE MAX(a,b) = (a > b) ? a : b;
DEFINE WIDTH(a) = CEIL(LOG2(a));

CONSTANT GROUP_ID = 7;
CONSTANT DEVICE_ID = 1;

CONSTANT ADDR_CONTROL      = h"0";
CONSTANT ADDR_VERSION      = h"1";
CONSTANT ADDR_ID           = h"2";
CONSTANT ADDR_LEDS         = h"3";
CONSTANT ADDR_RES4         = h"4";
CONSTANT ADDR_RES5         = h"5";
CONSTANT ADDR_RES6         = h"6";
CONSTANT ADDR_RES7         = h"7";
CONSTANT ADDR_DAC0         = h"8";
CONSTANT ADDR_DAC1         = h"9";
CONSTANT ADDR_DAC2         = h"A";
CONSTANT ADDR_DAC3         = h"B";
CONSTANT ADDR_RES12        = h"C";
CONSTANT ADDR_RES13        = h"D";
CONSTANT ADDR_RES14        = h"E";
CONSTANT ADDR_RES15        = h"F";

-- control register bits
CONSTANT CB_SCI_BYPASS      = 0;
CONSTANT CB_REF_SYNC_ENA   = 1;
CONSTANT CB_CLR_PDPINT_SRC = 2;
CONSTANT CB_SYNC_ENA       = 3;
CONSTANT CB_BREAK_ENA      = 4;
CONSTANT CB_IGBT_ERR_ENA   = 5;
CONSTANT CB_OC_ENA         = 6;

-- status register bits
CONSTANT SB_LINE_BREAK      = 0;
CONSTANT SB_LEM_OC         = 1;
CONSTANT SB_ERR_TOP0       = 2;
CONSTANT SB_ERR_BOT0       = 3;
CONSTANT SB_ERR_TOP1       = 4;
CONSTANT SB_ERR_BOT1       = 5;

```



```

CONSTANT STATUS_REG_WIDTH = 6;

-----

SUBDESIGN mdecode
(
    a[3..0] : INPUT = GND; -- DSP address 3 to 0
    w/r : INPUT = VCC; -- DSP write/read
    /we : INPUT = VCC; -- DSP write enable
    clk20 : INPUT = GND; -- DSP clock output

    /is : INPUT = VCC; -- I/O space

    d[7..0] : BIDIR = GND; -- bidirectional data bus

    xint1 : OUTPUT; -- DSP external interrupt 1
    /pdpint : OUTPUT; -- DSP power protect interrupt

    /reset : INPUT = VCC; -- board reset from/to DSP

    /dac_cs : OUTPUT; -- Quad-DAC chip select

    /dled[3..0] : OUTPUT; -- debug leds

    -- serial port (F240 SCI)
    xf : INPUT = GND; -- DSP external flag
    /bio : OUTPUT; -- DSP branch control input
    scirxd : OUTPUT; -- DSP SCI receive data
    scitxd : INPUT = GND; -- DSP SCI transmit data

    rs232_rx : OUTPUT; -- RS232 receive data
    rs232_tx : INPUT = GND; -- RS232 transmit data
    rs232_rts : INPUT = GND; -- RS232 request to send
    rs232_cts : OUTPUT; -- RS232 clear to send

    -- serial port (F240 SPI)
    spisimo : INPUT = GND; -- DSP SPI master data output
    spiste_sdi : INPUT = GND; -- SPI serial data in reg enable
    spiste_sdo : INPUT = GND; -- SPI serial data out reg enable
    spiste_sc : INPUT = GND; -- SPI status/control reg enable
    spiclk : INPUT = GND; -- SPI clock input

    spisomi : OUTPUT; -- DSP SPI master data input

    rx_sci : INPUT = GND; -- sci fiber data receive
    tx_sci : OUTPUT; -- sci fiber data transmit

    -- protection
    /e_top0 : INPUT = VCC; -- IGBT error feedback
    /e_bot0 : INPUT = VCC; -- IGBT error feedback
    /e_top1 : INPUT = VCC; -- IGBT error feedback
    /e_bot1 : INPUT = VCC; -- IGBT error feedback
    /lemoc : INPUT = VCC; -- Over current error

    -- SFL
    tx_clk : OUTPUT; -- serial clk decode out (use as ref sync gen)
    tx_sd : OUTPUT; -- serial data decode out (use as sync ena)
    rx_clk : INPUT = GND; -- serial clk decode in (use as carrier sync detect)
    rx_sd : INPUT = GND; -- serial data decode in (use as ref sync detect)
    scon : INPUT = GND; -- serial control decode in (use as line break detect)

    -- reserve unused pins
    res_tck : INPUT = GND;
    res_tdo : INPUT = GND;
    res_spiclk : INPUT = GND;
)

-----

VARIABLE

target_id_node[7..0] : NODE; -- node for group/device IDd
d_tri[7..0] : TRI_STATE_NODE; -- data output tri nodes

led_reg[3..0] : NODE; -- debug leds

sci_bypass : NODE; -- SCI bypass control bit

line_break : NODE;
clr_pdpint_src : NODE;

sync_ena : NODE;
break_ena : NODE;

status_reg[STATUS_REG_WIDTH-1..0] : NODE;

trig_pdpint : NODE;

ref_sync_ena : NODE;
ref_sync_det : NODE;
carrier_sync_det : NODE;

igbt_err_ena : NODE;
oc_ena : NODE;

pdpint_pulse_gen : pulse_et WITH (LENGTH = 7,
    TRIGGER_DEPTH = 1,
    TRIGGER_EDGE = "rising" );

xint1_pulse_gen : pulse_et WITH (LENGTH = 7,

```

```

TRIGGER_DEPTH = 1,
TRIGGER_EDGE = "rising" );

BEGIN

-----< misc >-----

-- data bus nodes
d[] = d_tri[];

-- decode DAC (address 0x8 to 0xB in I/O space)
/dac_cs = !(a[3..2] == 2) & !/is;

-----< LEDs >-----

FOR i IN 0 TO 3 GENERATE
    led_reg[i] = DFFE(d[i], clk20, /reset,, (a[3..0] == ADDR_leds) & !/is & !/we);
    /dled[i] = !led_reg[i];
END GENERATE;

-----< version & ID >-----

target_id_node[3..0] = DEVICE_ID;
target_id_node[7..4] = GROUP_ID;

FOR i IN 0 TO 7 GENERATE
    d_tri[i] = TRI(target_id_node[i], (a[3..0] == ADDR_ID) & !w/r & !/is);
END GENERATE;

-----< PDP interrupt >-----

-- generate status register
status_reg[SB_LINE_BREAK] = SRFF(line_break & !clr_pdpint_src, clr_pdpint_src, clk20, /reset,);
status_reg[SB_LEM_OC] = SRFF(!/lemoc & !clr_pdpint_src, clr_pdpint_src, clk20, /reset,);
status_reg[SB_ERR_TOP0] = SRFF(!/e_top0 & !clr_pdpint_src, clr_pdpint_src, clk20, /reset,);
status_reg[SB_ERR_BOT0] = SRFF(!/e_bot0 & !clr_pdpint_src, clr_pdpint_src, clk20, /reset,);
status_reg[SB_ERR_TOP1] = SRFF(!/e_top1 & !clr_pdpint_src, clr_pdpint_src, clk20, /reset,);
status_reg[SB_ERR_BOT1] = SRFF(!/e_bot1 & !clr_pdpint_src, clr_pdpint_src, clk20, /reset,);

FOR i IN 0 TO (STATUS_REG_WIDTH-1) GENERATE
    d_tri[i] = TRI(status_reg[i], (a[3..0] == ADDR_CONTROL) & !w/r & !/is);
END GENERATE;

-- trigger for pd interrupt
trig_pdpint = status_reg[SB_LINE_BREAK] & break_ena
              # status_reg[SB_LEM_OC] & oc_ena
              # status_reg[SB_ERR_TOP0] & igbt_err_ena
              # status_reg[SB_ERR_BOT0] & igbt_err_ena
              # status_reg[SB_ERR_TOP1] & igbt_err_ena
              # status_reg[SB_ERR_BOT1] & igbt_err_ena;

-- pd interrupt pulse generator
pdpint_pulse_gen.clk = clk20;
pdpint_pulse_gen.d = DFF(trig_pdpint, clk20,,);
pdpint_pulse_gen./reset = /reset;

-- pdpint src register clear signal
clr_pdpint_src = d[CB_CLR_PDPINT_SRC] & (a[3..0] == ADDR_CONTROL) & !/is & !/we;

-- pd interrupt signal
/pdpint = DFF(!pdpint_pulse_gen.q, clk20,,);

-- xint1 pulse generator
xint1_pulse_gen.clk = clk20;
xint1_pulse_gen.d = ref_sync_det;
xint1_pulse_gen./reset = /reset;

xint1 = DFF(!xint1_pulse_gen.q, clk20,,);

-----< SCI >-----

%
The SCI source can be the RS232 port OR the fiber optic interface.
The bypass bit determines if the output directly reflects the input.
%

-- SCI mode control register
sci_bypass = DFFE(d[CB_SCI_BYPASS], clk20,, /reset, (a[3..0] == ADDR_CONTROL) & !/is & !/we);

rs232_cts = GND; -- reserve

-- SCI mode AND rx signal polarity
CASE (sci_bypass) IS

-- fiber with bypass enabled
WHEN b"1" =>

    scirxd = rx_sci;
    tx_sci = rx_sci;
    rs232_rx = GND;

-- fiber with bypass disabled
WHEN b"0" =>

    scirxd = rx_sci;
    tx_sci = scitxd;
    rs232_rx = GND;

%

-- RS232
WHEN b"10" =>
    scirxd = rs232_tx;

```



```

        rs232_rx    = scitxd;
        tx_sci     = rx_sci;

-- disabled
    WHEN b"11" =>
        scirxd     = GND;
        rs232_rx    = GND;
        fsctx_do    = GND;

%

END CASE;

-----< SPI >-----

spisomi = TRI(GND, GND);

-----< SFL >-----

-- connect I/O pins
tx_clk = ref_sync_ena;
tx_sd = DFF(sync_ena & !trig_pdpint, clk20,,);
carrier_sync_det = DFF(rx_clk, clk20,,);
ref_sync_det = DFF(rx_sd, clk20,,);
line_break = DFF(scon, clk20,,);

-- control bits
ref_sync_ena = DFFE(d CB_REF_SYNC_ENA), clk20, /reset,, (a[3..0] == ADDR_CONTROL) & !/is & !/we);
sync_ena     = DFFE(d CB_SYNC_ENA),      clk20, /reset,, (a[3..0] == ADDR_CONTROL) & !/is & !/we);
break_ena    = DFFE(d CB_BREAK_ENA),      clk20, /reset,, (a[3..0] == ADDR_CONTROL) & !/is & !/we);
igbt_err_ena = DFFE(d CB_IGBT_ERR_ENA),   clk20, /reset,, (a[3..0] == ADDR_CONTROL) & !/is & !/we);
oc_ena       = DFFE(d CB_OC_ENA),         clk20, /reset,, (a[3..0] == ADDR_CONTROL) & !/is & !/we);

-- carrier sync detect output to dsp
/bio = DFF(carrier_sync_det, clk20, /reset,); -- also drives F240 CAP1 input

```

## **ModCon slave fiber EPLD main source file**

```

-----
END;

%          ----< ModCon Slave fiber epld >----
-----

file:      sfiber.tdf
project:   Module Controller (ModCon)
developer: J.A. du Toit
company:   University of Stellenbosch

-----

2001/08/15 (ver.0.2.0)
· Use fdil, fdol as sync pulse train

2001/05/28 (ver.0.1.0):
· Changed SFLMC10 rx data receive length to 8 bits

-----

%

INCLUDE "sflmc10a_dec.inc";

INCLUDE "lpm_counter.inc";
INCLUDE "edge_detect.inc";

INCLUDE "pulse_et.inc";
INCLUDE "pulse_lt.inc";

DEFINE MAX(a,b) = (a > b) ? a : b;
DEFINE WIDTH(a) = CEIL(LOG2(a));

CONSTANT SYNC_GEN_LENGTH = 16;
CONSTANT SYNC_SIG_LENGTH = 7;
CONSTANT SYNC_DET_LENGTH = 10;

CONSTANT BREAK_DET_LENGTH = 31;
CONSTANT BREAK_SIG_LENGTH = 15;

-----

SUBDESIGN sfiber
(
    r/w          : INPUT = VCC;      -- DSP read/write
    /strb        : INPUT = VCC;      -- DSP memory/IO strobe
    clk40        : INPUT = GND;      -- Main clock output
    /ds          : INPUT = VCC;      -- data space
    /ps          : INPUT = VCC;      -- program space

    /reset       : INPUT = VCC;      -- board reset from/to DSP

    /sram_cs     : OUTPUT;           -- sram chip select
    /sram_we     : OUTPUT;           -- sram write enable
    /sram_rd     : OUTPUT;           -- sram output enable

```

```

-- fiber optics
fdi[1..0] : INPUT = GND; -- input from optic fiber rx
fdo[1..0] : OUTPUT; -- output to optic fiber tx

-- SFL
tx_clk : INPUT = GND; -- serial receive clk (use as ref sync gen)
tx_sd : INPUT = GND; -- serial transmit data (use as sync ena)
rx_clk : OUTPUT; -- serial transmit clk (use as carrier sync detect)
rx_sd : OUTPUT; -- serial receive data (use as ref sync detect)
scon : OUTPUT; -- serial control out (use as line break detect)

-- delay signals
deli[1..0][2..0] : INPUT = GND; -- delay signals input
delo[1..0] : OUTPUT; -- delay signals output

-- SCI fiber
tx_sci : INPUT = GND; -- sci fiber transmit data
rx_sci : OUTPUT; -- sci fiber received data

-- reserve unused pins
res_f[2..0] : INPUT = GND;

master_sync : INPUT = VCC; -- sync from DSP
)

-----
VARIABLE

high_sync_pulse_det : pulse_lt WITH (LENGTH = SYNC_DET_LENGTH);
high_sync_pulse_sig : pulse_et WITH (LENGTH = SYNC_SIG_LENGTH,
                                     TRIGGER_DEPTH = 1,
                                     TRIGGER_EDGE = "falling" );

low_sync_pulse_det : pulse_lt WITH (LENGTH = SYNC_DET_LENGTH);
low_sync_pulse_sig : pulse_et WITH (LENGTH = SYNC_SIG_LENGTH,
                                     TRIGGER_DEPTH = 1,
                                     TRIGGER_EDGE = "falling" );

link_break_det : pulse_et WITH (LENGTH = BREAK_DET_LENGTH,
                                TRIGGER_DEPTH = 1,
                                TRIGGER_EDGE = "both" );

link_break_sig : pulse_lt WITH (LENGTH = BREAK_SIG_LENGTH);

sync_in : NODE;
sync_out : NODE;

ref_sync_ena : NODE;
sync_ena : NODE;
carrier_sync_det : NODE;
ref_sync_det : NODE;
line_break_det : NODE;

BEGIN

-----< misc >-----

-- external SRAM (decode in both program AND data space)
/sram_we = r/w # /strb;
/sram_rd = !r/w # /strb;
/sram_cs = /ds & /ps;

-----< SCI >-----

fdo0 = tx_sci; -- fiber optic out = SCI transmit
rx_sci = fdi0; -- SCI receive (inverted if first device in chain)

-----< SFL >-----

-- delay signals
delo0 = GND;
delo1 = GND;

-- connect I/O pins
ref_sync_ena = GND;
sync_ena = tx_sd;
rx_clk = carrier_sync_det;
rx_sd = ref_sync_det;
scon = link_break_sig.q;

fdo1 = DFF(sync_in & sync_ena, clk40,,);
sync_in = DFF(fdi1, clk40,,);

-- link break detect and signal pulse generators
link_break_det.clk = clk40;
link_break_det.d = sync_in;
link_break_det./reset = /reset;

line_break_det = !link_break_det.q;

link_break_sig.clk = clk40;
link_break_sig.d = !link_break_det.q;
link_break_sig./reset = /reset;

-- high sync pulse detector

```



END;

```

--...< ModCon Slave decode EPLD >...--

```

```
2001/08/21 (ver.0.4.0):
· generate separate carrier and reference sync pulses
· added enable bits to control register to select pdpint sources

2001/08/20 (ver.0.3.0):
· added control and status registers for protection
· reference sync out moved to xint

2001/08/15 (ver.0.2.0):
· Use fdil, fdol as sync pulse train

2001/06/01 (ver.0.1.0):
· Initial version
```

page 142

```

CONSTANT CB_BREAK_ENA      = 4;
CONSTANT CB_IGBT_ERR_ENA   = 5;
CONSTANT CB_OC_ENA         = 6;

-- status register bits
CONSTANT SB_LINE_BREAK     = 0;
CONSTANT SB_LEM_OC         = 1;
CONSTANT SB_ERR_TOP0       = 2;
CONSTANT SB_ERR_BOT0       = 3;
CONSTANT SB_ERR_TOP1       = 4;
CONSTANT SB_ERR_BOT1       = 5;

CONSTANT STATUS_REG_WIDTH  = 6;

-----

SUBDESIGN sdecode
(
    a[3..0]                : INPUT = GND;      -- DSP address 3 to 0
    w/r                    : INPUT = VCC;      -- DSP write/read
    /we                    : INPUT = VCC;      -- DSP write enable
    clk20                  : INPUT = GND;      -- DSP clock output

    /is                    : INPUT = VCC;      -- I/O space

    d[7..0]                : BIDIR = GND;      -- bidirectional data bus

    xint1                  : OUTPUT;           -- DSP external interrupt 1
    /pdpint                : OUTPUT;           -- DSP power protect interrupt

    /reset                 : INPUT = VCC;      -- board reset from/to DSP

    /dac_cs                : OUTPUT;           -- Quad-DAC chip select

    /dled[3..0]            : OUTPUT;           -- debug leds

    -- serial port (F240 SCI)
    xf                    : INPUT = GND;      -- DSP external flag
    /bio                   : OUTPUT;           -- DSP branch control input
    scirxd                 : OUTPUT;           -- DSP SCI receive data
    scitxd                 : INPUT = GND;      -- DSP SCI transmit data

    rs232_rx               : OUTPUT;           -- RS232 receive data
    rs232_tx               : INPUT = GND;      -- RS232 transmit data
    rs232_rts              : INPUT = GND;      -- RS232 request to send
    rs232_cts              : OUTPUT;           -- RS232 clear to send

    -- serial port (F240 SPI)
    spisimo                : INPUT = GND;      -- DSP SPI master data output
    spiste_sdi             : INPUT = GND;      -- SPI serial data in reg enable
    spiste_sdo             : INPUT = GND;      -- SPI serial data out reg enable
    spiste_sc              : INPUT = GND;      -- SPI status/control reg enable
    spiclk                 : INPUT = GND;      -- SPI clock input

    spisomi                : OUTPUT;           -- DSP SPI master data input

    rx_sci                 : INPUT = GND;      -- sci fiber data receive
    tx_sci                 : OUTPUT;           -- sci fiber data transmit

    -- protection
    /e_top0                : INPUT = VCC;      -- IGBT error feedback
    /e_bot0                : INPUT = VCC;      -- IGBT error feedback
    /e_top1                : INPUT = VCC;      -- IGBT error feedback
    /e_bot1                : INPUT = VCC;      -- IGBT error feedback
    /lemoc                 : INPUT = VCC;      -- Over current error

    -- SFL
    tx_clk                 : OUTPUT;           -- serial clk decode out (use as ref sync gen)
    tx_sd                  : OUTPUT;           -- serial data decode out (use as sync ena)
    rx_clk                 : INPUT = GND;      -- serial clk decode in (use as carrier sync detect)
    rx_sd                  : INPUT = GND;      -- serial data decode in (use as ref sync detect)
    scon                  : INPUT = GND;      -- serial control decode in (use as line break detect)

    -- reserve unused pins
    res_tck                : INPUT = GND;
    res_tdo                : INPUT = GND;
    res_spiclk             : INPUT = GND;
)

-----

VARIABLE

    target_id_node[7..0]   : NODE;           -- node for group/device IDd
    d_tri[7..0]            : TRI_STATE_NODE; -- data output tri nodes

    led_reg[3..0]          : NODE;           -- debug leds

    sci_bypass             : NODE;           -- SCI bypass control bit

    line_break             : NODE;
    clr_pdpint_src         : NODE;

    sync_ena               : NODE;
    break_ena              : NODE;

    status_reg[STATUS_REG_WIDTH-1..0] : NODE;

    trig_pdpint            : NODE;

```



```

ref_sync_ena          : NODE;
ref_sync_det          : NODE;
carrier_sync_det      : NODE;

igbt_err_ena          : NODE;
oc_ena                : NODE;

pdpint_pulse_gen      : pulse_et WITH (LENGTH = 7,
                                      TRIGGER_DEPTH = 1,
                                      TRIGGER_EDGE = "rising" );

xintl_pulse_gen        : pulse_et WITH (LENGTH = 7,
                                      TRIGGER_DEPTH = 1,
                                      TRIGGER_EDGE = "rising" );

BEGIN

-----< misc >-----

-- data bus nodes
d[] = d_tri[];

-- decode DAC (address 0x8 to 0xB in I/O space)
/dac_cs = !((a[3..2] == 2) & !/is);

-----< LEDs >-----

FOR i IN 0 TO 3 GENERATE
    led_reg[i] = DFFE(d[i], clk20, /reset,, (a[3..0] == ADDR_leds) & !/is & !/we);
    /dled[i] = !led_reg[i];
END GENERATE;

-----< version & ID >-----

target_id_node[3..0] = DEVICE_ID;
target_id_node[7..4] = GROUP_ID;

FOR i IN 0 TO 7 GENERATE
    d_tri[i] = TRI(target_id_node[i], (a[3..0] == ADDR_ID) & !w/r & !/is);
END GENERATE;

-----< PDP interrupt >-----

-- generate status register
status_reg[SB_LINE_BREAK] = SRFF(line_break & !clr_pdpint_src, clr_pdpint_src, clk20, /reset,);
status_reg[SB_LEM_OC]     = SRFF(!lemoc & !clr_pdpint_src, clr_pdpint_src, clk20, /reset,);
status_reg[SB_ERR_TOP0]   = SRFF(!e_top0 & !clr_pdpint_src, clr_pdpint_src, clk20, /reset,);
status_reg[SB_ERR_BOT0]   = SRFF(!e_bot0 & !clr_pdpint_src, clr_pdpint_src, clk20, /reset,);
status_reg[SB_ERR_TOP1]   = SRFF(!e_top1 & !clr_pdpint_src, clr_pdpint_src, clk20, /reset,);
status_reg[SB_ERR_BOT1]   = SRFF(!e_bot1 & !clr_pdpint_src, clr_pdpint_src, clk20, /reset,);

FOR i IN 0 TO (STATUS_REG_WIDTH-1) GENERATE
    d_tri[i] = TRI(status_reg[i], (a[3..0] == ADDR_CONTROL) & !w/r & !/is);
END GENERATE;

-- trigger for pd interrupt
trig_pdpint = status_reg[SB_LINE_BREAK] & break_ena
             # status_reg[SB_LEM_OC] & oc_ena
             # status_reg[SB_ERR_TOP0] & igbt_err_ena
             # status_reg[SB_ERR_BOT0] & igbt_err_ena
             # status_reg[SB_ERR_TOP1] & igbt_err_ena
             # status_reg[SB_ERR_BOT1] & igbt_err_ena;

-- pd interrupt pulse generator
pdpint_pulse_gen.clk = clk20;
pdpint_pulse_gen.d = DFF(trig_pdpint, clk20,,);
pdpint_pulse_gen./reset = /reset;

-- pdpint src register clear signal
clr_pdpint_src = d[CB_CLR_PDPINT_SRC] & (a[3..0] == ADDR_CONTROL) & !/is & !/we;

-- pd interrupt signal
/pdpint = DFF(!pdpint_pulse_gen.q, clk20,,);

-- xintl pulse generator
xintl_pulse_gen.clk = clk20;
xintl_pulse_gen.d = ref_sync_det;
xintl_pulse_gen./reset = /reset;

xintl = DFF(!xintl_pulse_gen.q, clk20,,);

-----< SCI >-----

%
The SCI source can be the RS232 port OR the fiber optic interface.
The bypass bit determines if the output directly reflects the input.
%

-- SCI mode control register
sci_bypass = DFFE(d[CB_SCI_BYPASS], clk20,, /reset, (a[3..0] == ADDR_CONTROL) & !/is & !/we);

rs232_cts = GND;    -- reserve

-- SCI mode AND rx signal polarity
CASE (sci_bypass) IS

-- fiber with bypass enabled
    WHEN b"1" =>
        scirxd = rx_sci;
        tx_sci = rx_sci;

```

```

        rs232_rx = GND;

-- fiber with bypass disabled
    WHEN b"0" =>
        scirxd = rx_sci;
        tx_sci = scitxd;
        rs232_rx = GND;

%
-- RS232
    WHEN b"10" =>
        scirxd      = rs232_tx;
        rs232_rx    = scitxd;
        tx_sci      = rx_sci;

-- disabled
    WHEN b"11" =>
        scirxd      = GND;
        rs232_rx    = GND;
        fsctx_do    = GND;

%

END CASE;

-----< SPI >-----

    spisomi = TRI(GND, GND);

-----< SFL >-----

-- connect I/O pins
tx_clk = ref_sync_ena;
tx_sd = DFF(sync_ena & !trig_pdpint, clk20,,);
carrier_sync_det = DFF(rx_clk, clk20,,);
ref_sync_det = DFF(rx_sd, clk20,,);
line_break = DFF(scon, clk20,,);

-- control bits
ref_sync_ena = DFFE(d CB_REF_SYNC_ENA], clk20, /reset,, (a[3..0] == ADDR_CONTROL) & !/is & !/we);
sync_ena     = DFFE(d CB_SYNC_ENA],      clk20, /reset,, (a[3..0] == ADDR_CONTROL) & !/is & !/we);
break_ena    = DFFE(d CB_BREAK_ENA],      clk20, /reset,, (a[3..0] == ADDR_CONTROL) & !/is & !/we);
igbt_err_ena = DFFE(d CB_IGBT_ERR_ENA],    clk20, /reset,, (a[3..0] == ADDR_CONTROL) & !/is & !/we);
oc_ena       = DFFE(d CB_OC_ENA],          clk20, /reset,, (a[3..0] == ADDR_CONTROL) & !/is & !/we);

-- carrier sync detect output to dsp
/bio = DFF(carrier_sync_det, clk20, /reset,); -- also drives F240 CAP1 input

-----

END;

```

## **ModCon edge detector source file**

```

%          ----< edge detector >----

-----

file:      edge_detect.tdf
project:   Module Controller (ModCon)
developer: J.A. du Toit
company:   University of Stellenbosch

-----

2001/06/01 (ver.1.0.0):
· Initial version

-----

%

PARAMETERS
(
    DEPTH = 2
);

ASSERT (DEPTH > 0)
REPORT      "DFF depth must be > 0"
SEVERITY    ERROR;

SUBDESIGN edge_detect
(
    d                : INPUT;
    clk              : INPUT;
    e                : OUTPUT;
    re               : OUTPUT;
    fe               : OUTPUT;
    q[DEPTH-1..0]   : OUTPUT;
)

% ----- %

VARIABLE

    q[DEPTH-1..0]   : DFF;

```



```

% ----- %
BEGIN

q[0].clk = clk;
q[0].d = d;

IF (DEPTH == 1) GENERATE

    re = d & !q[0].q;
    fe = !d & q[0].q;
    e = d $ q[0].q;

ELSE GENERATE

    FOR i IN 1 TO (DEPTH-1) GENERATE
        q[i].d = q[i-1].q;
    END GENERATE;

    re = q[(DEPTH-2)] & !q[(DEPTH-1)].q;
    fe = !q[(DEPTH-2)] & q[(DEPTH-1)].q;
    e = q[(DEPTH-2)] $ q[(DEPTH-1)].q;

END GENERATE;

% ----- %
END;

```

## **ModCon hysteresis trigger source file**

```

INCLUDE "lpm_counter.inc";
INCLUDE "edge_detect.inc";

PARAMETERS
(
    max_count      = 63,
    resync_clk_en  = "no"
);

DEFINE count_size() = CEIL(LOG2(max_count));

SUBDESIGN hyst_trigger
(
    d                : INPUT;
    clk              : INPUT;
    clk_en           : INPUT = vcc;
    level            : OUTPUT;
    pulse            : OUTPUT;
)

% ----- %

VARIABLE

q[(count_size()-1)..0] : NODE;
count_high              : NODE;
count_zero              : NODE;
direction               : NODE;
count_en                : NODE;

IF (USED(clk_en)) GENERATE
    clk_enable : NODE;
END GENERATE;

% ----- %

BEGIN

ASSERT (max_count > 2)
    REPORT      "max_count must be > 2"
    SEVERITY    ERROR;

ASSERT
    REPORT      "max_count value is % with counter width %" max_count, count_size()
    SEVERITY    INFO;

ASSERT (!(resync_clk_en == "yes"))
    REPORT      "re-synch clock enable input"
    SEVERITY    INFO;

IF (USED(clk_en)) GENERATE
    IF (resync_clk_en == "yes") GENERATE
        clk_enable = DFF(clk_en, clk,,);
    ELSE GENERATE
        clk_enable = clk_en;
    END GENERATE;
END GENERATE;

% counter direction up/down %
direction = DFF(d, clk,,);

% counter for hysteresis operation %
(q[(count_size()-1)..0],,,,,,,,,,,,,) = lpm_counter
(
    .clock = clk,

```

```

        .cnt_en = count_en,
        .updown = direction)
    WITH ( LPM_WIDTH = count_size());

% trigger levels %
count_high = q[] == max_count;
count_zero = q[] == 0;

% clamp maximum to max_count %
IF (USED(clk_en)) GENERATE
    count_en = clk_enable AND
               !(direction AND count_high) AND
               !(direction AND count_zero);
ELSE GENERATE
    count_en = !(direction AND count_high) AND
               !(direction AND count_zero);
END GENERATE;

% the output is set when the counter reaches max_count and cleared when
% the count reaches zero %
level = SRFF(count_high, count_zero, clk,,);
pulse = edge_detect(level, clk) WITH (level = "rising", depth = 2);

% ----- %

END;

```

## ModCon edge triggered pulse generator source file

```

%          ---< pulse generator >---
% -----

file:      pulse.tdf
project:   Module Controller (ModCon)
developer: J.A. du Toit
company:   University of Stellenbosch

% -----

2001/08/16 (ver.0.1.0):
· Initial version

% -----

%
DEFINE MAX(a,b) = (a > b) ? a : b;
DEFINE WIDTH(a) = CEIL(LOG2(a));

INCLUDE "lpm_counter.inc";
INCLUDE "edge_detect.inc";

PARAMETERS
(
    TRIGGER_EDGE = "both",
    TRIGGER_DEPTH = 2,
    LENGTH = 8
);

ASSERT (TRIGGER_DEPTH > 0)
REPORT "DFF depth must be > 0"
SEVERITY ERROR;

% -----

SUBDESIGN pulse
(
    d                                : INPUT;
    clk                              : INPUT;
    q                                : OUTPUT;
    /reset                           : INPUT;
)

% -----

VARIABLE

    pulse_counter                    : LPM_COUNTER WITH ( LPM_WIDTH = WIDTH(LENGTH+1),
                                                         LPM_DIRECTION = "down",
                                                         LPM_SVALUE = LENGTH);

    trigger                          : edge_detect WITH (DEPTH = TRIGGER_DEPTH);

    counter_busy                     : NODE;

% -----

BEGIN

    trigger.d = d;
    trigger.clk = clk;

    IF (TRIGGER_EDGE == "rising") GENERATE
        ASSERT REPORT "using rising edge" SEVERITY INFO;
        pulse_counter.sset = trigger.re;
    ELSE GENERATE

```



```

IF (TRIGGER_EDGE == "falling") GENERATE
  ASSERT REPORT "using falling edge" SEVERITY INFO;
  pulse_counter.sset = trigger.fe;
ELSE GENERATE
  ASSERT REPORT "using both edges" SEVERITY INFO;
  pulse_counter.sset = trigger.e;
END GENERATE;
END GENERATE;

counter_busy = pulse_counter.q[] != 0;

pulse_counter.cnt_en = counter_busy;
pulse_counter.clock = clk;
pulse_counter.aclr = !/reset;

q = counter_busy;

-----
END;

```

## **ModCon level triggered pulse generator source file**

```

%          ----< pulse generator level triggered >----
-----

file:      pulse_lt.tdf
project:   Module Controller (ModCon)
developer: J.A. du Toit
company:   University of Stellenbosch

-----

2001/08/16 (ver.0.1.0):
  Initial version

-----

%

DEFINE MAX(a,b) = (a > b) ? a : b;
DEFINE WIDTH(a) = CEIL(LOG2(a));

INCLUDE "lpm_counter.inc";
INCLUDE "edge_detect.inc";

PARAMETERS
(
  LENGTH = 8
);

-----

SUBDESIGN pulse_lt
(
  d          : INPUT;
  clk        : INPUT;
  q          : OUTPUT;
  /reset     : INPUT;
)

-----

VARIABLE

  pulse_counter      : LPM_COUNTER WITH ( LPM_WIDTH = WIDTH(LENGTH+1),
                                           LPM_DIRECTION = "down",
                                           LPM_SVALUE = LENGTH);

  counter_busy       : NODE;

-----

BEGIN

  pulse_counter.sset = d;

  counter_busy = pulse_counter.q[] != 0;

  pulse_counter.cnt_en = counter_busy;
  pulse_counter.clock = clk;
  pulse_counter.aclr = !/reset;

  q = counter_busy;

-----

END;

```

## **ModCon 4 bit serial CRC generator source file**

```

%          ----< 4 bit crc generator (crc_mc4) >----

```

```

-----
file:      crc_mc4.tdf
project:   Module Controller (ModCon)
developer: J.A. du Toit
company:   University of Stellenbosch
-----

2001/06/05 (ver.1.1.0):
  · Added normal shift operation (crc_sftn input)

2001/05/29 (ver.1.0.0):
  · Added crc GOOD_VALUE match output

2001/05/24 (ver.0.9.0):
  · Added initial value

2001/05/22 (ver.0.1.0):
  · Initial version
-----

%
PARAMETERS
(
    INITIAL_VALUE = h"5",
    GOOD_VALUE = h"5"
);

SUBDESIGN crc_mc4
(
    clk                : INPUT = gnd;      -- clock input
    sft_en             : INPUT = gnd;      -- shift enable
    sset               : INPUT = gnd;      -- synchronous set
    sdin               : INPUT = gnd;      -- serial data in
    aclr               : INPUT = gnd;      -- async reset input

    q[3..0]            : OUTPUT;           -- 4 bit crc out

    match              : OUTPUT;           -- crc matches GOOD_VALUE

    crc_sftn           : INPUT = VCC;      -- select shift or crc mode
    sdout              : OUTPUT;           -- msb out
)

-----

VARIABLE

-- CRC check (CRC-MC4)
crc_r[3..0]          : NODE;              -- crc registers
crc_x[4..0]          : NODE;              -- crc xor terms

BEGIN

-----

%
use CRC polynomial X4 + X3 + X0 (11001)
%

crc_x0 = crc_x4 & crc_sftn;
crc_x1 = crc_r0;
crc_x2 = crc_r1;
crc_x3 = crc_r2 $ crc_x4 & crc_sftn;
crc_x4 = crc_r3 $ sdin;

FOR i IN 0 TO 3 GENERATE
    IF (INITIAL_VALUE AND 2^i) GENERATE
        crc_r[i] = DFFE(crc_x[i] OR sset, clk, !aclr,, sft_en OR sset);
    ELSE GENERATE
        crc_r[i] = DFFE(crc_x[i] AND !sset, clk, !aclr,, sft_en OR sset);
    END GENERATE;
END GENERATE;

q[i] = crc_r[i];

match = crc_r[i] == GOOD_VALUE;

sdout = crc_r3;

-----

END;

```



# Appendix C

---

---

DSP software source

## C. DSP software source

The source listed in the appendix includes the source code for the master and slave module controllers. The code does not include header files defining DSP peripherals and registers.

The code is unified, except for the main source files, called master.c and slave.c.

The content of large tables were removed from the header files.

```

/*          -----< main >-----
-----

file:      main.c
project:   Module Controller (modcon)
developer: J.A. du Toit
company:   University of Stellenbosch
-----

2001/05/07 (ver.0.5.1):
Changed MCState and MCStage to enumerated values

2001/05/01 (ver.0.5.0):
moved RS232 code from interrupt routine to main loop
*/

/* ----- */
#include <stdlib.h>
#include <math.h>

#include "master.h"
#include "MC_lib.h"
#include "F240REGS.h"
#include "MC_RT1.h"
#include "MC_SCI.h"
#include "MC_ADC.h"
#include "MC_CMP.h"
#include "MC_EPLD.h"
#include "MC_PINS.h"
#include "MC_COMS.h"
#include "MC_SCI.h"

/* -----[ Global variables]----- */

/* const section must be at least 1 word for current boot/cmd environment */
const int dummy = 0;

/* ModCon operational info */
ULONG MCTime = 0;
TOPStage MCStage = OPSTAGE_SYNC_LOCK;
int MCTime = 0;
TOPState MCState = OPSTATE_UNDEF;

int SyncLocked = 0;

/* ModCon device IDs */
int TID;
int DID;
int GID;

/* references */
int Ref0 = 0;
int Ref1 = 0;
int Ref2 = 0;
int Ref3 = 0;

/* Status LEDs */
int LedSeqCnt = 0;

int LedSeq[8] = { 0xF, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 };

/* ADC measurement tables */
WORD ADC1Table[ADC_MAX_ENTRIES] = { 0, 0, 0, 0, 0, 0, 0, 0 };
WORD ADC2Table[ADC_MAX_ENTRIES] = { 0, 0, 0, 0, 0, 0, 0, 0 };

WORD ADCControlTable[ADC_MAX_ENTRIES] =
{
    adcSp5V | adcSSUPI | ADC1EN | ADC2EN | ADCIMSTART,
    adcSp5V | adcSSUPI | ADC1EN | ADC2EN | ADCIMSTART,
    adcSp5V | adcSSUPI | ADC1EN | ADC2EN | ADCIMSTART,
    adcSp5V | adcRefp25V | ADC1EN | ADC2EN | ADCIMSTART,
    adcSp5V | adcSp15V | ADC1EN | ADC2EN | ADCIMSTART,
    adcSp8V | adcSm15V | ADC1EN | ADC2EN | ADCIMSTART,
    adcMEAS_T1 | adcSp18VH | ADC1EN | ADC2EN | ADCIMSTART,
    adcMEAS_T0 | adcSm18VH | ADC1EN | ADC2EN | ADCIMSTART
};

```



```

int MeasVdcb[2] = {0, 0};
int MeasVout[2] = {0, 0};
int MeasIout[4] = {0, 0, 0, 0};

/* -----[ External variables] ----- */

/* RS232 SCI state machines */
extern TsciTxState SCITxState;
extern TsciState SCIState;
extern WORD SCIWDTCnt;

extern WORD iRxIn;
extern WORD iBuffIn;

/* ----- */

int main(void)
{
    int i;

    /* power on reset already handled by MC_res.asm routine (PLL setup etc.) */

    /* disable watchdog */
    iaWDCR = wdtCHK|wdtPS64|wdtDIS;

    /* initialize DSP registers and I/O pins */
    MC_Init();

    /* -----< check for illegal address reset >----- */

    if (iaSYSSR & BIT12)
    {
        iaSYSSR = iaSYSSR & 0xEFFF; /* clear reset status flag */
        MC_Shutdown(ERR_ILL_ADDR); /* Illegal address detected */
    }

    /* -----< check for watchdog timer reset >----- */

    if (iaSYSSR & BIT9)
    {
        iaSYSSR = iaSYSSR & 0xFDFF; /* clear reset status flag */

        /* Jump to Flash bootloader on WDT overflow */
        asm (" B 3E00h");
    }

    /* -----< check for software reset >----- */

    /* assume software reset was initiated by MC_Shutdown */

    if (iaSYSSR & BIT10)
    {
        iaSYSSR = iaSYSSR & 0xFBFF; /* clear reset status flag */

        /* Jump to Flash bootloader on WDT overflow */
        asm (" B 3E00h");
    }

    /*
    Timer1 generates a triangle waveform (up/down) with a period of 5kHz (timer prescale of 2).
    Timer2 is synchronized with timer1 and generates a sawtooth (up) waveform with a timer
    prescale of 1. Timer 2 will therefore match the period register 4 times as much as timer1.
    The timer1 period register and enable flag is used for timer2. This ensures synchronization
    between the timers. The timer2 period compare event is used to start the ADCs.

    Interrupts should therefore occur at n x 50us.

    The ADCs are started at each of these interrupts.

    ADC1:  0      m1(n)
          50      I_out
          100     m1(n+1)
          150     I_out

    ADC2:  0      Vdc
          50      m2(n)
          100     Vout
          150     m2(n+1)

    The supervisory code within the interrupt routine checks for activity on the host port
    before resetting the the WDT. This ensures that the ModCon returns to a known state
    when communication with the host is lost.

    */

    /* Initialize timers */
    GPT_Init();

    /* Initialize PWM compare and output logic */
    PWM_Init();

    /* Initialize capture units */
    CAP_Init();

    /* Initialize ADCs */
    ADC_Init();

    /* Initialize SCI */

```

```

SCI_Init(baud115200);

/* startup delay for master */
GPT_Delay(200);

/* initialize interrupts */
INT_Init();

/*
The MC stage must be synchronized with the PWM ramp. The interrupt routine
can then check to see if the stages progress as designed. If the stages drift
out of sync it means that false interrupts are generated or interrupts are missed.
This can be caused by interrupt routines that take too long to execute etc.

The sync is done by waiting for the timer1 period match (top of triangle).
Stage 2 is then entered. The interrupt is called directly after the global interrupt mask is cleared.
*/

/* Set stage and enable interrupts */
MCStage = OPSTAGE_SYNC_LOCK;
MCState = OPSTATE_WAIT_SYNC_LOCK;
iaEVIFRA = 0xFFFF; /* clear all flags */
asm (" CLRC INTM");

GPT_Delay(100);

/* From this stage the watchdog timer must be reset from within the timer1 interrupt */

/* enable watchdog timer with prescale 64 WDTCLK=32kHz => 500ms */
iaWDCR = wdtCHK|wdtPS64;
RTI_WDTKick();

/* start sync pulse generation */
iaCOMCON = COMCON_NORM_STOP;

/* Reset SCI state machines */
SCIState = SCI_IDLE;
SCITxState = SCI_TX_IDLE;

/* display IDs on debug leds */
LedSeq[LED_SEQ_DID] = DID;
LedSeq[LED_SEQ_GID] = GID;

/* main loop */
while (TRUE)
{
    /* Do SCI comms */
    SCI_DoComs();

    LedSeq[LED_SEQ_MC_STATE] = MCState;

    /* check MCState */
    if (MCState == OPSTATE_TIMEOUT_IDLE)
    {
        /* wait 2 seconds to ensure all modules trip */
        GPT_Delay(1000);
        SCIWDTCnt = 0;
        SCI_BuffReset();
        MCState = OPSTATE_NORM_IDLE;
    }
}

/* This code should never execute */
MC_Shutdown(ERR_END_OF_CODE);
}

/* ----- */

/* -----< main >----- */

-----

file:      main.c
project:   Module Controller (modcon)
developer: J.A. du Toit
company:   University of Stellenbosch

-----

2001/05/07 (ver.0.5.1):
Changed MCState and MCStage to enumerated values

2001/05/01 (ver.0.5.0):
moved RS232 code from interrupt routine to main loop

*/

/* ----- */

#include <stdlib.h>
#include <math.h>

#include "slave.h"
#include "MC_lib.h"
#include "F240REGS.h"
#include "MC_RTI.h"
#include "MC_SCI.h"
#include "MC_ADC.h"
#include "MC_CMP.h"
#include "MC_EPLD.h"

```



```

#include "MC_PINS.h"
#include "MC_COMS.h"
#include "MC_SCI.h"
#include "MC_PWM.h"

/* -----[ Global variables] ----- */

/* const section must be at least 1 word for current boot/cmd environment */
const int dummy = 0;

/* ModCon operational info */
ULONG MCTime = 0;
TOPStage MCStage = OPSTAGE_SYNC_LOCK;
int MCPhase = 0;
TOPState MCState = OPSTATE_UNDEF;

int SyncLocked = 0;

/* ModCon device ID */
int TID;
int DID;
int GID;

/* references */
int Ref[REF_COUNT_MAX] = {0, 0, 0, 0, 0, 0, 0, 0};
const int RefMin[REF_COUNT_MAX] = {-32767, 0, 0, PWM_REF_SIN_OL, 0, 0, 0, 0};
const int RefMax[REF_COUNT_MAX] = {+32767, 100, 10, PWM_REF_SINSQR_IREG, 0, 0, 0, 0};

/* Status LEDs */
int LedSeqCnt = 0;

int LedSeq[8] = {0xF, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0};

/* ADC measurement tables */
WORD ADC1Table[ADC_MAX_ENTRIES] = {0, 0, 0, 0, 0, 0, 0, 0};
WORD ADC2Table[ADC_MAX_ENTRIES] = {0, 0, 0, 0, 0, 0, 0, 0};

WORD ADCControlTable[ADC_MAX_ENTRIES] =
{
    adcSp5V | adcSSUPI | ADC1EN | ADC2EN | ADCIMSTART,
    adcSp5V | adcSSUPI | ADC1EN | ADC2EN | ADCIMSTART,
    adcSp5V | adcSSUPI | ADC1EN | ADC2EN | ADCIMSTART,
    adcSp5V | adcRefp25V | ADC1EN | ADC2EN | ADCIMSTART,
    adcSp5V | adcSp15V | ADC1EN | ADC2EN | ADCIMSTART,
    adcSp8V | adcSm15V | ADC1EN | ADC2EN | ADCIMSTART,
    adcMEAS_T1 | adcSp18VH | ADC1EN | ADC2EN | ADCIMSTART,
    adcMEAS_T0 | adcSm18VH | ADC1EN | ADC2EN | ADCIMSTART
};

int MeasVdcb[2] = {0, 0};
int MeasVout[2] = {0, 0};
int MeasIout[4] = {0, 0, 0, 0};

/* -----[ External variables] ----- */

/* RS232 SCI state machines */
extern TsciTxState SCITxState;
extern TsciState SCIState;
extern WORD SCIWDTCnt;

/* ----- */

int main(void)
{
    int i;

    /* power on reset already handled by MC_res.asm routine (PLL setup etc.) */

    /* disable watchdog */
    iaWDCR = wdtCHK|wdtPS64|wdtDIS;

    /* initialize DSP registers and I/O pins */
    MC_Init();

    /* -----< check for illegal address reset >----- */

    if (iaSYSSR & BIT12)
    {
        iaSYSSR = iaSYSSR & 0xEFFF; /* clear reset status flag */
        /*MC_Shutdown(errIllegalAddr); Illegal address detected */
    }

    /* -----< check for watchdog timer reset >----- */

    if (iaSYSSR & BIT9)
    {
        iaSYSSR = iaSYSSR & 0xFDFF; /* clear reset status flag */

        /* Jump to Flash bootloader on WDT overflow */
        asm (" B 3E00h");
    }

    /* -----< check for software reset >----- */

    /* assume software reset was initialized by MC_Shutdown */

    if (iaSYSSR & BIT10)
    {
        iaSYSSR = iaSYSSR & 0xFBFF; /* clear reset status flag */

        /* Jump to Flash bootloader on WDT overflow */
    }
}

```

```

    asm (" B 3E00h");
}

/*
Timer1 generates a triangle waveform (up/down) with a period of 5kHz (timer prescale of 2).
Timer2 is synchronized with timer1 and generates a sawtooth (up) waveform with a timer
prescale of 1. Timer 2 will therefore match the period register 4 times as much as timer1.
The timer1 period register and enable flag is used for timer2. This ensures synchronization
between the timers. The timer2 period compare event is used to start the ADCs.

Interrupts should therefore occur at n x 50us.

The ADCs are started at each of these interrupts.

ADC1:  0      m1(n)
        50     I_out
        100    m1(n+1)
        150    I_out

ADC2:  0      Vdc
        50     m2(n)
        100    Vout
        150    m2(n+1)

The supervisory code within the interrupt routine checks for activity on the host port
before resetting the WDT. This ensures that the ModCon returns to a known state
when communication with the host is lost.

*/

/* Initialize timers */
GPT_Init();

/* Initialize PWM compare and output logic */
PWM_Init();

/* Initialize capture units */
CAP_Init();

/* Initialize ADCs */
ADC_Init();

/* Initialize SCI */
SCI_Init(baud115200);

/* Startup delay for slave devices */
GPT_Delay(500);

/* initialize interrupts */
INT_Init();

/*
The MC stage must be synchronized with the PWM ramp. The interrupt routine
can then check to see if the stages progress as designed. If the stages drift
out of sync it means that false interrupts are generated or interrupts are missed.
This can be caused by interrupt routines that take too long to execute etc.

The sync is done by waiting for the timer1 period match (top of triangle).
Stage 2 is then entered. The interrupt is called directly after the global interrupt mask is cleared.

*/

/* Set stage and enable interrupts */
MCStage = OPSTAGE_SYNC_LOCK;
MCState = OPSTATE_WAIT_SYNC_LOCK;
iaEVIFRA = 0xFFFF; /* clear all flags */
asm (" CLRC INTM");

GPT_Delay(100);

/* From this stage the watchdog timer must be reset from within the timer1 interrupt */

/* enable watchdog timer with prescale 64 WDTCLK=32kHz => 500ms */
iaWDCR = wdtCHK|wdtPS64;
RTI_WDTKick();

/* start sync pulse generation */
iaCOMCON = COMCON_NORM_STOP;

/* Reset SCI state machines */
SCIState = SCI_IDLE;
SCITxState = SCI_TX_IDLE;

/* display IDs on debug leds */
LedSeq(LED_SEQ_DID) = DID;
LedSeq(LED_SEQ_GID) = GID;

/* main loop */
while (TRUE)
{
    /* Do SCI comms */
    SCI_DoComs();

    /* check MCState */
    if (MCState == OPSTATE_TIMEOUT_IDLE)
    {
        /* wait 2 seconds to ensure all modules trip */
        GPT_Delay(1000);
        SCIWDTCnt = 0;
    }
}

```



```

        SCI_BuffReset();
        MCState = OPSTATE_NORM_IDLE;
    }

}

/* This code should never execute */
MC_Shutdown(ERR_END_OF_CODE);
}

/* ----- */
/* -----< ModCon Interrupts >----- */
/* ----- */

file:      MC_INT.c
project:   Module Controller (modcon)
developer: J.A. du Toit
company:   University of Stellenbosch

-----

2001/05/07 (ver.0.5.1):
Changed MCStage to enumerated values

2001/05/01 (ver.0.5.0):
moved RS232 code to main loop

2001/04/18 (ver.0.4.0):
Changed the async transparency from RS232 to IrDA (DLE/STX, DLE/ETX to SOF, EOF)
*/

/* ----- */

#ifdef _MC_MASTER
#include "master.h"
#endif

#ifdef _MC_SLAVE
#include "slave.h"
#endif

#include "F240REGS.h"
#include "MC_INT.h"
#include "MC_LIB.h"
#include "MC_RTI.h"
#include "MC_ADC.h"
#include "MC_SCI.h"
#include "MC_EPLD.h"
#include "MC_GPT.h"
#include "MC_CMP.h"
#include "MC_COMS.h"
#include "MC_err.h"
#include "MC_asm.h"
#include "MC_CAP.h"
#include "MC_DACQ.h"
#include "MC_PWM.h"

/* ----- */

#define SYNC_ERR_MAX    4
#define SYNC_ERR_MIN    -4
#define SYNC_REF        T1PERIOD_NOM/2

#define LINE_STATUS_VALID_COUNT 100

/* -----[ Global variables] ----- */

int IntErrCnt;

StageTime Stage0Time = { 0, 0, 0, 0 };
StageTime Stage1Time = { 0, 0, 0, 0 };
StageTime Stage2Time = { 0, 0, 0, 0 };
StageTime Stage3Time = { 0, 0, 0, 0 };

int DacqFlags = 0;
int DacqSrcCnt = 0;
int *DacqDst = DACQ_ADDR_START;
int DacqSrc[ DACQ_SRC_MAXNUM ];

TRefType RefType = PWM_REF_SIN_OL;

/* -----[ External variables] ----- */

/* ModCon operational info */
extern TopStage MCStage;
extern TopState MCState;
extern int  MCPhase;

extern ULONG MCTime;

/* ADC tables and measurements */
extern WORD ADC1Table[];
extern WORD ADC2Table[];
extern WORD ADCControlTable[];

extern int MeasVdcb[];
extern int MeasVout[];
extern int MeasIout[];

```

```

/* LED states */
extern int LedSeqCnt;
extern int LedSeq[];

/* references */
extern int Ref[];

/* SCI comms */
extern WORD SCIWDTCnt;
extern TsciTxState SCITxState;
extern TsciState SCIState;

/* ----- */

void INT_Init(void)
{
    /*
        INT1: None
        INT2: Timer 2 period interrupt (issued 4 times per switching cycle)
        INT3: None
        INT5: None
        INT6: None

        Interrupts should be enabled after calling this routine.
    */

    asm (" SETC INTM");

    /* setup interrupts */
    iaIFR = 0xFFFF; /* clear all flags */
    iaIMR = 0x0006; /* enable int3 (EV group B) */

    /* EV group A source = power drive protect */
    iaEVIFRA = 0xFFFF;
    iaEVIMRA = 0x0001;

    /* EV group B source = timer2 period */
    iaEVIFRB = 0xFFFF;
    iaEVIMRB = 0x0001;

    /* EV group C source = none */
    iaEVIFRC = 0xFFFF;
    iaEVIMRC = 0x0000;

    /* XINT1 interrupt (disabled, low priority, poll for results) */
    iaXINT1CR = XINT1_PRI_LOW | XINT1_EDGE_RIS;

    /* clear false interrupt counter */
    IntErrCnt = 0;
}

/* ----- */

interrupt void Int_2()
{
    /*
        EV Group A interrupt
    */

    volatile WORD Int2Vec = iaEVIVRA;
    int epld_status;
    int pdpint_error;

    /* check if interrupt source power drive protect */
    if (Int2Vec == PDPINTVEC)
    {
        epld_status = EPLD_RegRead(EPLD_CONTROL);

        /*
            The cause of the PDPINT can be determined by reading the EPLD status register

            The order of priority is:
            [ 1] line break
            [ 2] IGBT error
            [ 3] LEMM over current
        */

        /* check if IGBT error */
        if (epld_status & (SB_ERR_TOP0|SB_ERR_BOT0|SB_ERR_TOP1|SB_ERR_BOT1))
        {
            pdpint_error = ERR_IGBT_GROUP;

            if (epld_status & SB_ERR_TOP0) pdpint_error |= ERR_IGBT_TOP0;
            if (epld_status & SB_ERR_BOT0) pdpint_error |= ERR_IGBT_BOT0;
            if (epld_status & SB_ERR_TOP1) pdpint_error |= ERR_IGBT_TOP1;
            if (epld_status & SB_ERR_BOT1) pdpint_error |= ERR_IGBT_BOT1;

            MC_Shutdown(pdpint_error);
        }

        /* check if lem over current */
        else if (epld_status & SB_LEM_OC) MC_Shutdown(ERR_OC_AC_HARD);

        /* check if line break detected */
        else if (epld_status & SB_LINE_BREAK) MC_Shutdown(ERR_BREAK_DET);

        else MC_Shutdown(ERR_INV_PDPINT_SRC);
    }
}

```



```

    }

    else IntErrCnt++;
}

/* ----- */

interrupt void Int_3()
{
/*
    EV Group B interrupt

    The 200us period is divided into 4 stages of 50us each. The stages are numbered 0 to 3.
    The following events occur during each stage:

    Stage 0:   Check stage synchronization. This is done by checking the state of timer1.
               At this time, timer1 should be near zero and counting up (BIT13 of GPTCON).
               Set ADCs to convert MEAS_I and MEAS_V0

    Stage 1:   If asymmetric PWM is used, the duty cycles are calculated and updated.

    Stage 2:   Set ADCs to convert MEAS_I and MEAS_V1

    Stage 3:   The PWM duty cycles are calculated and updated.

    Note:      The first interrupt will enter stage 2 (after initial stage increment).
               The first time stage 3 is entered, the ADCControlTableIndex is incremented (0 to 1).
*/

static int ADCIndx = 0;
static int SyncErrAcc;
static int SyncErr;

extern int SyncLocked;
extern int MCPhase;

volatile WORD Int3Vec = iaEVIVRB;

int epld_status;
int dummy;

/* check if interrupt source is timer2 period match */
if (Int3Vec == T2PINTVEC)
{
    switch (MCStage)
    {
/* ----- Stage 0 ----- */
        case OPSTAGE_STAGE0:
        {
            /* store Stage entry time */
            Stage0Time.entry = iaT2CNT;

            /* read ADC values from FIFOs */
            ADC1Table[ADCIndx] = iaADCFIFO1>>6;
            ADC2Table[ADCIndx] = iaADCFIFO2>>6;

            /* select next set of measurements and start conversion */
            iaADCTRL1 = adcMEAS_I|adcMEAS_V1|ADC1EN|ADC2EN|ADCIMSTART;

            /* check stage synchronization (must be counting up and count<256) */
            if ((iaT1CNT & 0xFF00) || (~iaGPTCON & BIT13)) MC_Shutdown(ERR_STAGE_SYNC);

#ifdef _MC_SLAVE
            /* restore nominal Timer1&2 Periods */
            iaT1PR = T1PERIOD_NOM;
            iaT2PR = T1PERIOD_NOM-1;

            /* check to see if sync capture event occurred during stage3 */
            if (iaCAPFIFO & CAP1FIFO_MASK)
            {
                /* determine error and set SyncLocked variable */
                SyncErr = SYNC_REF - iaCAP1FIFO;
                SyncLocked = SYNC_LOCK_TIMEOUT;

                if (SyncErr < SYNC_ERR_MIN)
                {
                    iaT1PR = T1PERIOD_NOM+1;
                    iaT2PR = T1PERIOD_NOM;
                }
                else if (SyncErr > SYNC_ERR_MAX)
                {
                    iaT1PR = T1PERIOD_NOM-1;
                    iaT2PR = T1PERIOD_NOM-2;
                }
            }
            else
            {
                if (--SyncLocked == 0) MC_Shutdown(ERR_SYNC_LOST);
            }

            /* increment the MCTime variable (200us increments) */
            MCTime++;

```

```

/* increment phase count (0 to 199 in 100us increments), should be even in stage 0 */
if (++MCPhase >= MC_PHASE_MAX) MCPhase = 0;

#endif

#ifdef MC_MASTER
/* increment the MCTime variable (200us increments) */
MCTime++;

/* increment phase count (0 to 199 in 100us increments), should be even in stage 0 */
if (++MCPhase >= MC_PHASE_MAX)
{
    MCPhase = 0;
    /* send reference sync pulse */
    EPLD_RegWrite(EPLD_CONTROL, EPLD_CONTROL_RUN|CB_REF_SYNC_ENA);
}
else
{
    EPLD_RegWrite(EPLD_CONTROL, EPLD_CONTROL_RUN);
}
}

#endif

if (!(MCTime & 0x0FFF))
{
    /* update led state */
    if (++LedSeqCnt > 7) LedSeqCnt = 0;
    EPLD_RegWrite(EPLD_LEDS, LedSeqCnt);
}

/* select next ADC channels */
if (++ADCIndx >= ADC_MAX_ENTRIES) ADCIndx = 0;

/* SCI transmit/receive check */
SCI_DoTx();
SCI_DoRx();

/* wait for ADC EOC and store previous measurements */
while (iaADCTRL1 & ADCEOC);

/* store Stage time */
Stage0Time.EOC = iaT2CNT;

/* store ADC values */
MeasIout[0] = iaADCIF01>>6;
MeasVdcb[0] = iaADCIF02>>6;

/* select next set of measurements and start conversion */
iaADCTRL1 = ADCControlTable[ADCIndx];

/* data acquisition */
DoDACQ();

/* next stage */
MCStage = OPSTAGE_STAGE1;

/* store Stage exit time */
/* Stage0Time.exit = iaT2CNT; */
dummy = iaT2CNT;
if (dummy > Stage0Time.exit) Stage0Time.exit = dummy;

break;
}

/* ----- Stage 1 ----- */
case OPSTAGE_STAGE1:
{
    /* store Stage entry time */
    Stage1Time.entry = iaT2CNT;

    /* read ADC values from FIFOs */
    ADC1Table[ADCIndx] = iaADCIF01>>6;
    ADC2Table[ADCIndx] = iaADCIF02>>6;

    /* select next set of measurements and start conversion */
    iaADCTRL1 = adcMEAS_I|adcMEAS_V0|ADC1EN|ADC2EN|ADCIMSTART;

    /* select next ADC channels */
    if (++ADCIndx >= ADC_MAX_ENTRIES) ADCIndx = 0;

    /* SCI transmit/receive check */
    SCI_DoTx();
    SCI_DoRx();

    /* wait for ADC EOC and store previous measurements */
    while (iaADCTRL1 & ADCEOC);

    /* store Stage time */
    Stage1Time.EOC = iaT2CNT;

    /* store ADC values */
    MeasIout[1] = iaADCIF01>>6;
    MeasVout[0] = iaADCIF02>>6;

#ifdef MC_SLAVE
    /* update the PWM reference values */
    DoPWM();
#endif

    /* store Stage time */

```



```

    Stage1Time.PWM = iaT2CNT;

    /* select next set of measurements and start conversion */
    iaADCTRL1 = ADCControlTable[ADCIndx];

    /* data acquisition */
    DoDACQ();

    /* next stage */
    MCStage = OPSTAGE_STAGE2;

    /* store Stage exit time */
    /* Stage1Time.exit = iaT2CNT; */
    dummy = iaT2CNT;
    if (dummy > Stage1Time.exit) Stage1Time.exit = dummy;

    break;
}

/* ----- Stage 2 ----- */
case OPSTAGE_STAGE2:
{
    /* store Stage entry time */
    Stage2Time.entry = iaT2CNT;

    /* read ADC values from FIFOs */
    ADC1Table[ADCIndx] = iaADCFIFO1>>6;
    ADC2Table[ADCIndx] = iaADCFIFO2>>6;

    /* select next set of measurements and start conversion */
    iaADCTRL1 = adcMEAS_I|adcMEAS_V1|ADC1EN|ADC2EN|ADCIMSTART;

#ifdef MC_SLAVE
    /* check for reference sync from master */
    if (iaXINT1CR & XINT1_FLAG)
    {
        /* reset phase counter if set */
        MCPhase = 1;

        /* set trigger for DACQ */
        DacqFlags |= DACQ_START_TRIG;

        /* clear flag (only needed if interrupt not used) */
        iaXINT1CR = XINT1_PRI_LOW | XINT1_EDGE_RIS;
    }
    /* increment phase count */
    else MCPhase++;
#endif

#ifdef MC_MASTER
    /* increment phase variable */
    MCPhase++;

    /* increment SCI WDT counter and check timeout */
    if (MCState == OPSTATE_NORM_RUN)
    {
        if (++SCIWDTCnt == SCI_WDT_COUNT_MAX) MC_Shutdown(ERR_SCI_NR_TIMEOUT);
    }
#endif

    /* select next ADC channels */
    if (++ADCIndx >= ADC_MAX_ENTRIES) ADCIndx = 0;

    /* SCI transmit/receive check */
    SCI_DoTx();
    SCI_DoRx();

    /* wait for ADC EOC and store previous measurements */
    while (iaADCTRL1 & ADCEOC);

    /* store Stage time */
    Stage2Time.EOC = iaT2CNT;

    /* store ADC values */
    MeasIout[2] = iaADCFIFO1>>6;
    MeasVdcb[1] = iaADCFIFO2>>6;

    /* select next set of measurements and start conversion */
    iaADCTRL1 = ADCControlTable[ADCIndx];

    /* data acquisition */
    DoDACQ();

    /* next stage */
    MCStage = OPSTAGE_STAGE3;

    /* store Stage exit time */
    /* Stage2Time.exit = iaT2CNT; */
    dummy = iaT2CNT;
    if (dummy > Stage2Time.exit) Stage2Time.exit = dummy;

    break;
}

/* ----- Stage 3 ----- */
case OPSTAGE_STAGE3:
{

```

```

/* store Stage entry time */
Stage3Time.entry = iaT2CNT;

/* read ADC values from FIFOs */
ADC1Table[ADCIndx] = iaADC_FIFO1>>6;
ADC2Table[ADCIndx] = iaADC_FIFO2>>6;

/* select next set of measurements and start conversion */
iaADCTRL1 = adcMEAS_I|adcMEAS_V0|ADC1EN|ADC2EN|ADCIMSTART;

/* clear CAP1 FIFO to test for sync event */
iaCAP_FIFO = CAP_FIFO_CLR;

/* select next ADC channels */
if (++ADCIndx >= ADC_MAX_ENTRIES) ADCIndx = 0;

/* SCI transmit/receive check */
SCI_DoTx();
SCI_DoRx();

/* wait for ADC EOC and store previous measurements */
while (iaADCTRL1 & ADCEOC);

/* store Stage time */
Stage3Time.EOC = iaT2CNT;

/* store ADC values */
MeasIout[3] = iaADC_FIFO1>>6;
MeasVout[1] = iaADC_FIFO2>>6;

#ifdef _MC_SLAVE
/* Update the PWM reference values */
DoPWM();
#endif

/* store Stage time */
Stage3Time.PWM = iaT2CNT;

/* select next set of measurements and start conversion */
iaADCTRL1 = ADCControlTable[ADCIndx];

/* data acquisition */
DoDAQ();

/* next stage */
MCStage = OPSTAGE_STAGE0;

/* store Stage exit time */
/* Stage3Time.exit = iaT2CNT; */
dummy = iaT2CNT;
if (dummy > Stage3Time.exit) Stage3Time.exit = dummy;

break;
}

/* ----- Sync Lock ----- */
case OPSTAGE_SYNC_LOCK:
{
#ifdef _MC_MASTER
/*
The master remains in this stage until the line_break signal on the input
clears and remains cleared for LINE_STATUS_VALID_COUNT. This implies that all
the slave modules have locked onto the sync pulse train (slave devices enable their
sync output after lock).
*/

epld_status = EPLD_RegRead(EPLD_CONTROL);

/* EPLD_RegWrite(EPLD_LEDS, MCStage); */
EPLD_RegWrite(EPLD_LEDS, epld_status);

/* check line break status */
if (epld_status & SB_LINE_BREAK)
{
/* line break, reset line valid status counter */
SyncLocked = 0;

/* clear pd protect int sources */
EPLD_RegWrite(EPLD_CONTROL, EPLD_CONTROL_WAIT_SYNC_CLR);
}
else
{
if (++SyncLocked >= LINE_STATUS_VALID_COUNT)
{
/* valid line status detected */
MCStage = OPSTAGE_STAGE_SYNC;
SyncLocked = SYNC_LOCK_TIMEOUT;

/* clear pd protect int sources and enable break detect */
EPLD_RegWrite(EPLD_CONTROL, EPLD_CONTROL_WAIT_SYNC_DONE);
}
}

/* SCI transmit/receive check */
SCI_DoTx();
SCI_DoRx();
}

#endif

#ifdef _MC_SLAVE

```



```

/*
    This stage is used to lock onto the incoming sync pulse train.

    The Timer1 & 2 periods are shortened to force the timers to catch up

    Timer1 values are then captured until the error is within 10 percent
    of half the Timer Period register value. The period is then reset
    to the nominal value and the next stage is entered. It is assumed
    that the phase will not drift more than approx 10% while in the
    next stage (max 3 interrupt cycles)
*/

iaT1PR = T1PERIOD_NOM+1;
iaT2PR = T1PERIOD_NOM;

/* clear pdpint sources, sync out disabled */
EPLD_RegWrite(EPLD_CONTROL, EPLD_CONTROL_WAIT_SYNC_CLR);

/* set led state */
EPLD_RegWrite(EPLD_LEDS, MCStage);

/* check to see if sync capture event */
if (iaCAPFIFO & CAP1FIFO_MASK)
{
    SyncErr = abs(iaCAP1FIFO - SYNC_REF);

    if ((SyncErr < 10) && (iaT1CNT < 200) && (iaGPTCON & T1STAT))
    {
        iaT1PR = T1PERIOD_NOM;
        iaT2PR = T1PERIOD_NOM-1;
        MCStage = OPSTAGE_STAGE_SYNC;
        SyncLocked = SYNC_LOCK_TIMEOUT;

        /* clear pd protect int sources and enable break detect and sync out */
        EPLD_RegWrite(EPLD_CONTROL, EPLD_CONTROL_WAIT_SYNC_DONE);
    }
}

#endif

break;
}

/* ----- Stage Sync ----- */
case OPSTAGE_STAGE_SYNC:
{
#ifdef _MC_MASTER
    /*
        Stay in this stage until stages are synched with PWM ramp
        This stage would be equivalent to Stage0, except that the next
        stage is not selected until the ramp conditions are met.
    */

    /* set led state */
    EPLD_RegWrite(EPLD_LEDS, MCStage);

    /* check stage synchronization (counting down and count<256) */
    if ((iaT1CNT & 0xFF00) || (~iaGPTCON & BIT13))
    {
        MCState = OPSTATE_WAIT_STAGE_SYNC;
    }
    else
    {
        /* ramp conditions met */

        /* wait for ADC end of conversion */
        while (iaADCTRL1 & ADCEOC);

        /* Clear ADC FIFO buffers (2 dummy reads) */
        ADCIdx = iaADCFIFO1;
        ADCIdx = iaADCFIFO1;
        ADCIdx = iaADCFIFO2;
        ADCIdx = iaADCFIFO2;

        /* set ADCs to convert MEAS_I and MEAS_V1 */
        ADCIdx = 1;
        iaADCTRL1 = ADCCControlTable[ADCIdx];

        /* Reset MCTime and MCPPhase variables */
        MCTime = 0;
        MCPPhase = 0;

        /* next stage */
        MCStage = OPSTAGE_STAGE1;
        MCState = OPSTATE_NORM_IDLE;

        SyncLocked = SYNC_LOCK_TIMEOUT;
    }
}
#endif

#ifdef _MC_SLAVE
    /*
        Stay in this stage until stages are synched with PWM ramp
        This stage would be equivalent to Stage0, except that the next
        stage is not selected until the ramp conditions are met.
    */

```

```

/* clear CAP1 FIFO to test for sync event */
iaCAPFIFO = CAPFIFO_CLR;

/* set led state */
EPLD_RegWrite(EPLD_LEDS, MCStage);

/* check stage synchronization (counting down and count<256) */
if ((iaT1CNT & 0xFF00) || (~iaGPTCON & BIT13))
{
    /* not yet, make sure that the PWM outputs are disabled */
    iaCOMCON = COMCON_NORM_STOP;
    MCState = OPSTATE_WAIT_STAGE_SYNC;
}
else
{
    /* ramp conditions met */

    /* wait for ADC end of conversion */
    while (iaADCTRL1 & ADCEOC);

    /* Clear ADC FIFO buffers (2 dummy reads) */
    ADCIndx = iaADCFIFO1;
    ADCIndx = iaADCFIFO1;
    ADCIndx = iaADCFIFO2;
    ADCIndx = iaADCFIFO2;

    /* set ADCs to convert MEAS_I and MEAS_V1 */
    ADCIndx = 1;
    iaADCTRL1 = ADCControlTable[ADCIndx];

    /* Reset MCTime and MCPhase variables */
    MCTime = 0;
    MCPhase = 0;

    /* next stage */
    MCStage = OPSTATE_STAGE1;
    MCState = OPSTATE_NORM_IDLE;

    SyncLocked = SYNC_LOCK_TIMEOUT;
}
}

#endif

break;
}

/* ----- */
default:
{
    /* This code should never execute. If entered, do shutdown */
    MC_Shutdown(ERR_INV_STAGE);

    break;
}
}

/* reset watchdog timer */
RTI_WDTKick();
}

else IntErrCnt++;
}

/* ----- */
/* -----< ModCon library >-----
-----

file:      MC_LIB.c
project:   Module Controller (ModCon)
developer: J.A. du Toit
company:   University of Stellenbosch

-----

2001/11/22 (ver.0.5.0)

2001/04/24 (ver.0.4.0):
moved some global variables to main.h
modified MC_Shutdown to initialize timers and display error on leds

2001/04/24 (ver.0.3.0):
Read device ID from EPLD

*/

/* ----- */

#define _MC_LIB_BUILD

#include <string.h>
#include <ioports.h>

#ifdef _MC_MASTER
#include "master.h"
#endif

#ifdef _MC_SLAVE
#include "slave.h"

```



```

#endif

#include "MC_LIB.h"
#include "F240REGS.h"
#include "MC_ADC.h"
#include "MC_GPT.h"
#include "MC_RTI.h"
#include "MC_PINS.h"
#include "MC_SCI.h"
#include "MC_EPLD.h"
#include "MC_CMP.h"
#include "MC_ASM.h"
#include "MC_INT.h"
#include "MC_ERR.h"
#include "MC_COMS.h"
#include "MC_CAP.h"
#include "MC_DACQ.h"

/* -----[ Global variables] ----- */

/* -----[ External variables] ----- */

extern int TID;
extern int DID;
extern int GID;

/* ----- */

void MC_Init(void)
{
    WORD i;
    WORD *Data = (WORD*)0x8000;
    WORD InfoLength;
    WORD InfoStringLength;
    WORD InfoCount;
    char InfoChar;

    /* gobally disable maskable interrupts */
    asm (" SETC INTM");

    /* Configure I/O pins (see MC_PINS.h file for details) */
    iaSCIPC2 = defSCIPC2;
    iaSPIPC1 = defSPIPC1;
    iaSPIPC2 = defSPIPC2;
    iaNMICR = defNMICR;
    iaXINT1CR = defXINT1CR;
    iaXINT2CR = defXINT2CR;
    iaXINT3CR = defXINT3CR;
    iaOCRA = defOCRA;
    iaOCRB = defOCRB;
    iaPADATDIR = defPADATDIR;
    iaPBDATDIR = defPBDATDIR;
    iaPCDATDIR = defPCDATDIR;

    /* read the device ID embedded in the EPLD (4 bit register) */
    TID = EPLD_RegRead(EPLD_TID) & 0x7F;
    GID = TID>4;
    DID = TID & 0x0F;

    /* Clear external SRAM (data space) */
    for (i=0x0000; i<0x8000; i++) *Data++ = 0;

    /*
    Copy target info to external memory (data space)

    0x8000 InfoLength
    0x8001 InfoCount
    0x8002 InfoIndex (start, length) * 62
    0x8080 InfoTable (null terminated strings of max 32 chars) * 62
    */

    InfoLength = GetInfoLength();
    /* check length of info struct, else truncate */
    if (InfoLength>(INFO_COUNT_MAX * INFO_SLENGTH_MAX)) InfoLength = (INFO_COUNT_MAX * INFO_SLENGTH_MAX);
    InfoCount = 0;

    *(WORD*)(INFO_INDEX_ADDR) = (WORD)(INFO_TABLE_ADDR);

    for (i=0; i<InfoLength; i++)
    {
        /* get character */
        InfoChar = GetInfoChar(i);
        InfoStringLength++;

        /* store in info table */
        *(char*)(INFO_TABLE_ADDR+i) = InfoChar;

        /* check if end of info string */
        if (InfoChar == chNUL)
        {
            *(WORD*)(INFO_INDEX_ADDR+1+2*InfoCount) = InfoStringLength;

            if (++InfoCount == 62) break;
            else *(WORD*)(INFO_INDEX_ADDR+2*InfoCount) = (WORD)(INFO_TABLE_ADDR+i+1);

            InfoStringLength = 0;
        }
    }
}

```

```

* (WORD*) (INFO_LENGTH_ADDR) = InfoLength;
* (WORD*) (INFO_COUNT_ADDR) = InfoCount;

/* clear data acquisition array */
for (i=0; i<DACQ_SRC_MAXNUM; i++) DacqSrc[i] = 0;
}

/* ----- */

void GPT_Init(void)
{
    /*
        Assume CPUCLK=20MHz

        Timer1:
        Set to 5kHz (taking into account up/down counting mode, Period = TPRD*2).
        Set to continuous up/down mode (for PWM generation).
        Set prescale to 1 (clock divide by two).

        Timer2:
        Use timer1 enable.
        Set to continuous up counting mode (Period = TPRD+1).
        Set prescale value of 0 (use input clock directly).
        Timer should run at 20kHz.
        Set timer2 to start ADC conversion (period interrupt flag).

        Timer3:
        Used for delay routines.
        Set to single up counting mode.
        Use own period register and enable flag.
        Set counting period to 1ms.
    */

    /* setup timer1 with timer disabled, setup timers 2 and 3, enable timer1 */

    /* Configure timer1 */
    iaT1CON = TMODECud|TPSd2;
    iaT1PR = T1PERIOD_NOM;
    iaT1CMPR = T1PERIOD_NOM/2;
    iaT1CNT = 0xFFFE;

    /* Configure timer2 */
    iaT2CON = TMODECu|TPSd1|TSWT1;
    iaT2PR = T1PERIOD_NOM-1;
    iaT2CMPR = T1PERIOD_NOM/2;
    iaT2CNT = 0xFFFE;

    /* Configure timer3 (1ms period) used for delays etc. */
    iaT3CON = TMODESu|TPSd32;
    iaT3PR = 625;
    iaT3CMPR = 0;
    iaT3CNT = 0xFFFE;

    /* start timer1 */
    iaT1CON = TMODECud|TPSd2|TENABLE;

    /* enable compare outputs */
    iaGPTCON = 0x046A;
}

/* ----- */

void PWM_Init(void)
{
    /*
        Blanking time of 5us
        Assume CPUCLK=20MHz
        Set prescale value to 1 for deadband and pwm counters (clock divide by two)

        PWM assignments (Full bridge)
        PWM1 -> right-top      (active high)
        PWM2 -> right-bottom   (active low)
        PWM3 -> left-top       (active high)
        PWM4 -> left-bottom    (active low)
        PWM5,6 unused

    */

    /* Configure compare registers for PWM */

    /* PWM1,3 active low. PWM2,4 active high, PWM5,6 forced low (Hold) */
    iaACTR = 0x0099;

    /* Deadband prescale x/2, enable deadband units, DB=5us */
    iaDBTCN = 0x32E8;

    /* set to duty cycle to 50% */
    iaCMPR1 = T1PERIOD_NOM/2;
    iaCMPR2 = T1PERIOD_NOM/2;
    iaCMPR3 = 0;

    /* PWM7 active high, PWM8,9 forced low (Hold) */
    iaSACTR = 2;

    iaSCMPR1 = T1PERIOD_NOM/2;
    iaSCMPR2 = 0;
    iaSCMPR3 = 0;
}

```



```

/*
    select CMP1-2 PWM mode,
    disable simple compare
    ACTCR reload on T1 underflow and period match
    Compare reload on underflow and period match
    Compare enable CENABLE
    Full compare outputs not enabled FCOMPOE
    Simple compare outputs enabled SCOMPOE
*/
iaCOMCON = resCOMCON;
iaCOMCON = defCOMCON;
}

/* ----- */
void CAP_Init(void)
{
    /*
        Setup capture units

        Use Timer2 as the base for CAP1
        Disable CAP2, CAP3 and CAP4
    */

    iaCAPCON = CAP1EDGE_RIS|CAPQEPN_ENA12|CAPRES;
}

/* ----- */
void GPT_Delay(WORD msec)
{
    WORD i;

    /* Timer3 is set to generate a millisecond period */
    for (i=0; i<msec; i++)
    {
        iaT3CON = TMODEsu|TPSd32|TENABLE; /* start timer */
        while (iaT3CON & TENABLE); /* wait for period match */
    }
}

/* ----- */
void MC_Shutdown(TF240Error error)
{
    int j;

    /* disable interrupts */
    asm (" SETC INTM");

    /* disable watchdog */
    iaWDCR = wdtCHK|wdtPS64|wdtDIS;

    /* Disable PWM outputs */
    iaCOMCON = COMCON_SHUTDOWN;

    /* disable sync pulse train */
    EPLD_RegWrite(EPLD_CONTROL, EPLD_CONTROL_SHUTDOWN);

    /* initialize timers (Timer3 used for delay function) */
    GPT_Init();

    for (j=0; j<4; j++)
    {
        EPLD_LEDSpin(15);
        EPLD_LEDSpin(-15);
    }
    /* display error on leds */
    for (j=0; j<4; j++)
    {
        EPLD_RegWrite(EPLD_LEDS, error>>4);
        GPT_Delay(1000);

        EPLD_LEDSpin(15);

        EPLD_RegWrite(EPLD_LEDS, error);
        GPT_Delay(500);

        EPLD_LEDSpin(-15);
    }

    /* set led state */
    EPLD_RegWrite(EPLD_LEDS, 0x00);

    GPT_Delay(1000);

    /* set led state */
    EPLD_RegWrite(EPLD_LEDS, 0xFF);

    /* Initiate software reset */
    iaSYSCR = iaSYSCR & 0x00FF;
}

```

```

    GPT_Delay(1000);

    /* this code should never execute */
    while (TRUE);
}

/* ----- */

void ADC_Init(void)
{
    /*
     * enable ADC1 and ADC2
     * set prescale to 12 => 6.0us @ SYSCLK=10MHz
     * enable event manager start of conversion
     */
    iaADCTRL1 = adcMEAS_I|adcMEAS_V0|ADC1EN|ADC2EN;
    iaADCTRL2 = ADCPSCALE;
}

/* ----- */

void EPLD_LEDSpin(int Speed)
{
    /* cycle leds, a positive Speed setting cycles left, negative right */

    if (Speed > 0)
    {
        EPLD_RegWrite(EPLD_LEDS, 1);
        GPT_Delay(Speed);
        EPLD_RegWrite(EPLD_LEDS, 2);
        GPT_Delay(Speed);
        EPLD_RegWrite(EPLD_LEDS, 4);
        GPT_Delay(Speed);
        EPLD_RegWrite(EPLD_LEDS, 8);
        GPT_Delay(Speed);
        EPLD_RegWrite(EPLD_LEDS, 0);
        GPT_Delay(Speed);
    }
    else
    {
        Speed = abs(Speed);
        EPLD_RegWrite(EPLD_LEDS, 8);
        GPT_Delay(Speed);
        EPLD_RegWrite(EPLD_LEDS, 4);
        GPT_Delay(Speed);
        EPLD_RegWrite(EPLD_LEDS, 2);
        GPT_Delay(Speed);
        EPLD_RegWrite(EPLD_LEDS, 1);
        GPT_Delay(Speed);
        EPLD_RegWrite(EPLD_LEDS, 0);
        GPT_Delay(Speed);
    }
}

/* ----- */
/* -----< ModCon RS232 communications >----- */
/* ----- */

file:      MC_SCI.c
project:   ModCon
developer: J.A. du Toit
company:   University of Stellenbosch

/* ----- */

2001/07/13 (ver.0.5.1)
fixed read bytes state machine (decoding failed after escape chars)

2001/07/03 (ver.0.5.0)

2001/05/08 (ver.0.4.5)
Added SCI transmit code

2001/05/07 (ver.0.4.0)
Added receiver error checks (parity, line break, framing and overflow)

*/

/* ----- */

#ifdef MC_MASTER
#include "master.h"
#endif

#ifdef MC_SLAVE
#include "slave.h"
#endif

#include "MC_SCI.h"
#include "MC_EPLD.h"
#include "MC_ERR.h"
#include "MC_COMS.h"
#include "F240REGS.h"
#include "MC_RTI.h"
#include "MC_GPT.h"
#include "MC_ADC.h"
#include "MC_CMP.h"
#include "MC_LIB.h"

```



```

#include "MC_DACQ.h"

/* -----[ Global variables] ----- */

/* receive errors */
TSCIErrors SCIErrors = { 0, 0, 0, 0, 0, 0 };

/* SCI receiver/transmitter buffer */
char SCIBuff[ SCI_BUFF_LENGTH ];
char RxBuff[ SCI_RXBUFF_LENGTH ];
char SCITxDBuff = 0xFF;

int SCI_CRC = 0;

WORD SCIWDTCnt = 0;
TsciState SCIState = SCI_IDLE;
TsciTxState SCITxState = SCI_TX_IDLE;

/* receive buffer indexes */
WORD iRxIn = 0;
WORD iRxOut = 0;

/* SCI buffer indexes */
WORD iBuffIn = 0;
WORD iBuffOut = 0;

WORD nBuffOut = 0;

int TempInt = 0;

/* -----[ External variables] ----- */

extern int MeasVdcb;
extern int MeasVout;
extern int MeasIout;

extern int MCPhase;
extern TOPStage MCStage;
extern TOPState MCState;

extern int TID;
extern int DID;
extern int GID;

extern LedSeq[];

extern int Ref[];
extern const int RefMax[];
extern const int RefMin[];

extern int LedSeqCnt;
extern ULONG MCTime;

/* ----- */

void SCI_Init(TbaudRate baud)
{
    /*
     * Initialize the SCI port of the F240
     * In the TI compatability mode, the baud rate is set to 38400 for communication
     * with the TI sci download software.
     *
     * The sci source can be the fiber optic or RS232
     *
     * The default baudrate is 38400 for TI compatability
     *
     * BRR = (SYSCLK/(baud*8))-1
     */

    /* Initialize SCI registers */

    /* 1 stop bit, no parity, idle line, 8 bits (0x17) */
    iaSCICCR = defSCICCR;

    /* enable TX and RX, internal SCICLK, disable RX ERR, SLEEP and TXWAKE (0x13) */
    iaSCICTL1 = defSCICTL1r;

    /* disable TX INT and RX/BK INT (0x00) */
    iaSCICTL2 = defSCICTL2;

    /* set BRR high and low bytes (0x00) */
    iaSCILBAUD = baud & 0xFF;
    iaSCIHBAUD = (baud>>8) & 0xFF;

    /* enable TXD and RXD pins (0x22) */
    iaSCIPC2 = 0x22;

    /* release SCI software reset (0x33) */
    iaSCICTL1 = defSCICTL1;
}

/* ----- */

int SCI_ProcessPkt(void)
{
    /*
     * The command consists of the following fields field:
     * CMD[ 7] R/W
     */
}

```

```

CMD[ 6..5]   Length
CMD[ 4..0]   CMD_ID

```

The length field translates as follows (include TID, CMD and CRC)

```

0 : 0 extra words      => pkt length = 3 bytes
1 : 1 extra word       => pkt length = 5 bytes
1 : 2 extra word       => pkt length = 7 bytes
3 : DLength extra words => pkt length = 4 + 2*DLength bytes

```

The order of processing is as follows:

- [ 1] Determine if device is destination target
- [ 2] Check packet length
- [ 3] Assemble CMD buffer and pass to SCI\_ProcessCmd function

iBuffIn points to the next empty space in the receive buffer => number of bytes in received ExPkt

The packet is processed as follows:

- [ 1] Check CRC:
  - \* The SCI\_CRC contains the checksum (including the transmitted checksum)
  - \* TxCRC = SCIBuff[ iBuffIn-1]
  - => CRC OK if ((SCI\_CRC - TxCRC) & 0xFF) = TxCRC

```
*/
```

```
WORD data0;
WORD data1;
```

```
int IDMatch;
TComsCmd CMD;
```

```
int TxCRC;
```

```
/* Check CRC */
TxCRC = SCIBuff[ iBuffIn-1];
```

```
/* continue if CRC OK */
if ((SCI_CRC - TxCRC) & 0xFF) == TxCRC)
```

```
{
    /* check if module is target */
    IDMatch = SCI_CheckTID(SCIBuff[ SCI_BUFF_TID_INDEX]);
```

```
    if (IDMatch)
```

```
    {
        /* read command */
        CMD = SCIBuff[ SCI_BUFF_CMD_INDEX];
```

```
        /* can only return data if device is only target */
        if ((CMD & SCI_RW_MASK) && (IDMatch != TARGET_ONLY))
```

```
        {
            SCIState = SCI_IDLE;
            return FALSE;
```

```
        }
        else
```

```
        {
            /* check packet length */
            switch (SCI_GET_LENGTH_FIELD(CMD))
```

```
            {
                case 0:
                {
                    if (iBuffIn == SCI_CMD_LENGTH_0)
                    {
                        SCI_ProcessCmd0(CMD);
                    }
                    else
                    {
                        /* length mismatch */
                        SCIErrors.LEN++;
                        SCIState = SCI_IDLE;
                        return FALSE;
                    }
                }
                break;
            }

```

```
            case 1:
            {
                if (iBuffIn == SCI_CMD_LENGTH_1)
                {
                    data0 = SCIBuff[ SCI_BUFF_DATA0_INDEX] | (SCIBuff[ SCI_BUFF_DATA0_INDEX+1] << 8);
                    SCI_ProcessCmd1(CMD, data0);
                }
                else
                {
                    /* length mismatch */
                    SCIErrors.LEN++;
                    SCIState = SCI_IDLE;
                    return FALSE;
                }
            }
            break;

```

```
            case 2:
            {
                if (iBuffIn == SCI_CMD_LENGTH_2)
                {
                    data0 = SCIBuff[ SCI_BUFF_DATA0_INDEX] | (SCIBuff[ SCI_BUFF_DATA0_INDEX+1] << 8);
                    data1 = SCIBuff[ SCI_BUFF_DATA1_INDEX] | (SCIBuff[ SCI_BUFF_DATA1_INDEX+1] << 8);
                    SCI_ProcessCmd2(CMD, data0, data1);
                }
                else
                {
                    /* length mismatch */
                    SCIErrors.LEN++;

```



```

        SCIState = SCI_IDLE;
        return FALSE;
    }
    break;
}

case 3:
{
    if (iBuffIn == SCI_CMD_LENGTH_3 + SCIBuff[SCI_BUFF_DLEN_INDEX])
    {
        SCI_ProcessCmd3(CMD, (BYTE)SCIBuff[SCI_BUFF_DLEN_INDEX],
            &SCIBuff[SCI_BUFF_DLEN_INDEX+1]);
    }
    else
    {
        /* length mismatch */
        SCIErrors.LEN++;
        SCIState = SCI_IDLE;
        return FALSE;
    }
    break;
}

default:
{
    SCIState = SCI_IDLE;
    return FALSE;
    break;
}
}
}
}
}
else
{
    /* CRC error, reset state machine */
    SCIErrors.CRC++;
    SCIState = SCI_IDLE;
    return FALSE;
}
}

/* ----- */

int SCI_ProcessCmd0(TComsCmd CMD)
{
    switch (CMD)
    {
        case CMD_NULL:
        {
            /* NULL cmd => do nothing */
            SCIState = SCI_IDLE;
            return TRUE;
            break;
        }

        case CMD_SHUTDOWN:
        {
            iaCOMCON = COMCON_SHUTDOWN; /* disable PWM signals */
            MC_Shutdown(ERR_HOST_REQ); /* Shutdown module */
            break;
        }

        case CMD_START:
        {
            #ifdef MC_SLAVE
            /* enable PWM signals */
            iaCOMCON = COMCON_NORM_RUN;
            #endif
            SCIState = SCI_IDLE;
            MCState = OPSTATE_NORM_RUN;
            return TRUE;
            break;
        }

        case CMD_STOP:
        {
            MCState = OPSTATE_NORM_IDLE;
            SCIState = SCI_IDLE;
            #ifdef MC_SLAVE
            iaCOMCON = COMCON_NORM_STOP; /* disable PWM signals */
            #endif
            return TRUE;
            break;
        }

        case CMD_SLEEP:
        {
            /* enter OPSTATE_SLEEP state if current state is OPSTATE_NORM_IDLE */
            if (MCState == OPSTATE_NORM_IDLE)
            {
                MCState = OPSTATE_SLEEP;
                SCIState = SCI_SLEEP;
                return TRUE;
            }
            else
            {
                SCIState = SCI_IDLE;
                return FALSE;
            }
        }
    }
}

```

```

        break;
    }

    case CMD_SYNC:
    {
        LedSeqCnt = 0;
        MCTime = 0;
        SCIState = SCI_IDLE;
        EPLD_RegWrite(EPLD_LEDS, LedSeq, LedSeqCnt);
        return TRUE;
        break;
    }

    case CMD_RD_STATE:
    {
        return SCI_SendWA((int*)&MCState, 1, TRUE);
        break;
    }

    case CMD_RD_STAGE:
    {
        return SCI_SendWA((int*)&MCStage, 1, TRUE);
        break;
    }
}

#ifdef _MC_SLAVE
    case CMD_RD_INFO_CNT:
    {
        return SCI_SendWA((int*)(INFO_COUNT_ADDR), 1, TRUE);
        break;
    }
#endif

#ifdef _MC_MASTER
    case CMD_RD_INFO_CNT:
    {
        TempInt = 0;
        return SCI_SendWA(&TempInt, 1, TRUE);
        break;
    }
#endif

    case CMD_DACQ_SINGLE:
    {
        /* clear busy and trigger flags and set start flag */
        DacqFlags = DACQ_START;
        SCIState = SCI_IDLE;
        return TRUE;
        break;
    }

    default:
    {
        /* undefined command */
        SCIState = SCI_IDLE;
        return FALSE;
        break;
    }
}

/* ----- */

int SCI_ProcessCmd1(TComsCmd CMD, WORD data)
{
    switch(CMD)
    {
        case CMD_RD_WDS:
        {
            return SCI_SendWA((int*)data, 1, TRUE);
            break;
        }

        case CMD_RD_REF:
        {
            /* data = ref# */
            if (data < REF_COUNT_MAX)
            {
                return SCI_SendWA((int*)&Ref[data], 1, TRUE);
            }
            else
            {
                SCIState = SCI_IDLE;
                return FALSE;
            }
            break;
        }

#ifdef _MC_SLAVE
        case CMD_RD_INFO_STR:
        {
            return SCI_SendInfoLine(data);
            break;
        }
#endif

        default:
        {

```



```

        return FALSE;
        break;
    }
}

/* ----- */
int SCI_ProcessCmd2(TComsCmd CMD, WORD data0, WORD data1)
{
    int CRC;

    switch (CMD)
    {
        case CMD_WR_WDS:
        {
            /* write a word to data space */
            if (IsValidDSAddr((int*)data0))
            {
                *(WORD*)data0 = data1;
                return TRUE;
            }
            else
            {
                SCIState = SCI_IDLE;
                return FALSE;
            }
        }

        case CMD_RD_WADS:
        {
            return SCI_SendWA((int*)data0, data1, TRUE);
            break;
        }

        case CMD_WR_REF:
        {
            /* data0 = ref#, data1 = new value */
            if (data0 < REF_COUNT_MAX)
            {
                Ref[data0] = data1;
                SCIState = SCI_IDLE;
                return TRUE;
            }
            else
            {
                SCIState = SCI_IDLE;
                return FALSE;
            }
            break;
        }

        case CMD_WR_REF_BNDS:
        {
            /* data0 = ref#, data1 = new value */
            if (data0 < REF_COUNT_MAX)
            {
                if (data1 > RefMax[data0])
                {
                    SCIState = SCI_IDLE;
                    return FALSE;
                }
                else if (data1 < RefMin[data0])
                {
                    SCIState = SCI_IDLE;
                    return FALSE;
                }
                else
                {
                    Ref[data0] = data1;
                    SCIState = SCI_IDLE;
                    return TRUE;
                }
            }
            else
            {
                SCIState = SCI_IDLE;
                return FALSE;
            }
            break;
        }

        case CMD_CALC_CRC:
        {
            /* calculate the CRC for data pointed to by D1, length of D0 */
            CRC = CalcCRC((char*)data0, data1);
            return SCI_SendWA(&CRC, 1, TRUE);
            break;
        }

        default:
        {
            SCIState = SCI_IDLE;
            return FALSE;
            break;
        }
    }
}

/* ----- */
int SCI_ProcessCmd3(TComsCmd CMD, BYTE DLength, char *data)
{

```

```

int i;
WORD temp;

switch (CMD)
{
    case CMD_DACQ_SELECT:
    {
        /* check if max number of sources exceeded */
        if (DLength > DACQ_SRC_MAXNUM*2)
        {
            SCISState = SCI_IDLE;
            return FALSE;
        }
        else
        {
            DacqFlags = 0;
            DacqSrcCnt = DLength/2;
            for (i=0; i<DacqSrcCnt; i++)
            {
                temp = *data++;
                temp |= (*data++)<<8;
                DacqSrc[i] = temp;
            }
            SCISState = SCI_IDLE;
            return TRUE;
        }
        break;
    }

    default:
    {
        SCISState = SCI_IDLE;
        return FALSE;
        break;
    }
}

}

/* ----- */

inline int SCI_CheckTID(int ID)
{
    /*
     * returns 0 if device is not destination
     * returns 1 if device is only destination
     * returns 2 if device is group destination
     * returns 4 if device is broadcast destination
     */

    if ((ID & 0x80) == 0) return TARGET_ALL; /* broadcast flag set */

    else if ((ID >> 4) == GID) /* groups IDs match */
    {
        if ((ID & 0x0F) == DID) return TARGET_ONLY; /* group and device IDs match */
        else if ((ID & 0x0F) == 0) return TARGET_GROUP; /* group IDs match, device ID = 0 */
    }

    return TARGET_NO;
}

/* ----- */

int SCI_SendWA(int *addr, int num, int sign)
{
    /*
     * Transmit a packet filled with num words of data
     * num is the number of word to send => bytes is 2 x num
     * the packet length field is the number of words-1
     *
     * The stored data is not masked because only the lower 8 bits are written to the SCI Tx register
     */

    #define MAX_NUM_OF_WORDS    SCI_NUM_OF_DATA_BYTES/2

    int i;
    int CRC = 0;
    int desc;

    char *Buff = SCIBuff;
    WORD data;

    if ((num >= 1) && (num <= MAX_NUM_OF_WORDS))
    {
        if (sign)
        {
            /* treat as signed integers */
            desc = (num-1) | DATA_TYPE_INT_ARRAY;
        }
        else
        {
            /* treat as unsigned integers */
            desc = (num-1) | DATA_TYPE_UINT_ARRAY;
        }

        *Buff++ = HID;
        CRC += HID;
    }
}

```



```

*Buff++ = TID;
CRC += TID;

*Buff++ = desc;
CRC += desc;

for (i=0; i<num; i++)
{
    /* check if valid addr, else substitute data with 0xDEAD */
    if (IsValidDSAddr(addr))
    {
        data = *addr;

        *Buff++ = data;
        CRC += data;

        *Buff++ = data>>8;
        CRC += data>>8;
    }
    else
    {
        data = 0xDEAD;

        *Buff++ = data;
        CRC += data;

        *Buff++ = data>>8;
        CRC += data>>8;
    }

    /* select next address */
    addr++;
}

/* append CRC */
*Buff++ = (CRC & 0xFF);

/* set number of characters to transmit */
nBuffOut = 2*num + TH_PKT_NUM_CONTROL_BYTES;

/* set state machine to transmit BOF */
SCITxState = SCI_TX_BOF;

return TRUE;
}
else
{
    SCISState = SCI_IDLE;
    return FALSE;
}
}

/* ----- */
int SCI_SendCA(char *addr, WORD num)
{
    /*
    Transmit a packet filled with num characters
    num is the number of characters to send
    the packet length field is the number of characters-1
    */

#define MAX_NUM_OF_CHARS    SCI_NUM_OF_DATA_BYTES

    int i;
    int CRC = 0;
    int desc;

    char *Buff = SCIBuff;
    WORD data;

    if ((num >= 1) && (num <= MAX_NUM_OF_CHARS))
    {
        desc = (num-1) | DATA_TYPE_CHAR_ARRAY;

        *Buff++ = HID;
        CRC += HID;

        *Buff++ = TID;
        CRC += TID;

        *Buff++ = desc;
        CRC += desc;

        for (i=0; i<num; i++)
        {
            /* check if valid addr, else substitute data with NULL */
            if (IsValidDSAddr(addr))
            {
                data = *addr;

                *Buff++ = data;
                CRC += data;
            }
            else

```

```

        {
            *Buff++ = chNUL;
        }

        /* select next address */
        addr++;
    }

    /* append CRC */
    *Buff++ = CRC & 0xFF;

    /* set number of characters to transmit */
    nBuffOut = num + TH_PKT_NUM_CONTROL_BYTES;

    /* set state machine to transmit BOF */
    SCITxState = SCI_TX_BOF;

    return TRUE;
}
else
{
    SCIState = SCI_IDLE;
    return FALSE;
}
}

/* ----- */

int SCI_SendInfoLine(WORD Line)
{
    /*
    0x8000 InfoLength
    0x8001 InfoCount
    0x8002 InfoIndex (start, length) * 62
    0x8040 InfoTable (null terminated strings of max 32 chars) * 62
    */

    WORD num;
    char* addr;

    /* check index */
    if (Line <= (WORD*)(INFO_COUNT_ADDR))
    {
        addr = (char*)(WORD*)(INFO_INDEX_ADDR+2*Line);
        num = *(WORD*)(INFO_INDEX_ADDR+1+2*Line);

        /* check address */
        if ((addr >= (char*)INFO_TABLE_ADDR) && (addr < (char*)(INFO_TABLE_ADDR + (INFO_COUNT_MAX-1)*INFO_SLENGTH_MAX)))
        {
            return SCI_SendCA(addr, num);
        }
        else return FALSE;
    }
    else return FALSE;
}

/* ----- */

void SCI_ProcessRxBytes(int Num)
{
    /*
    Read Num bytes from input buffer and decode

    The possible states are:
    Idle
    Busy
    BusyCE
    EOF
    Sleep
    Bypass
    */

    int i;
    char data;

    for (i=0; i<Num; i++)
    {
        /* get next character and decode */
        data = SCI_GetRxChar();

        /* A BOF character always signals the start of a new frame */
        if (data == chBOF)
        {
            iBuffIn = 0;
            SCIState = SCI_BUSY;
            SCI_CRC = 0;
        }
        else
        {
            switch (SCIState)
            {
                case SCI_BUSY:
                {
                    /* special cases are EOF and CE */
                    if (data == chCE)
                    {
                        SCIState = SCI_BUSY_CE;
                    }
                }
            }
        }
    }
}

```



```

        else if (data == chEOF)
        {
            SCIState = SCI_EOF;
            return;
        }
        else
        {
            /* add character and check for buffer overflow */
            if(!SCI_AddChar(data))
            {
                SCIErrors.OF++;
                SCIState = SCI_IDLE;
            }
        }
        break;
    }

    case SCI_BUSY_CE:
    {
        /* add character xor 0x20 and check for buffer overflow */
        if(!SCI_AddChar(data^0x20))
        {
            SCIErrors.OF++;
            SCIState = SCI_IDLE;
        }
        else
        {
            SCIState = SCI_BUSY;
        }
        break;
    }

    case SCI_SLEEP:
    {
        break;
    }

    default:
    {
        break;
    }
}
}
}

/* ----- */
/*          ---< ModCon main header file >---          */
/* ----- */

file:      main.h
project:   ModCon
developer: J.A. du Toit
company:   University of Stellenbosch

-----

2001/05/07 (ver.0.4.0):
2001/05/01 (ver.0.3.0):
*/

#ifndef _MASTER
#define _MASTER

/* ----- */

#include "MC_TYPES.h"
#include "MC_ERR.h"

/* ----- */

#define MC_PHASE_MAX      199

/* ----- */

#define LED_SEQ_START      0
#define LED_SEQ_DID        1
#define LED_SEQ_GID        2
#define LED_SEQ_MC_STATE   3
#define LED_SEQ_RES1       4
#define LED_SEQ_RES2       5
#define LED_SEQ_RES3       6
#define LED_SEQ_END        7

/* ----- */

#endif

/*          ---< ModCon main header file >---          */
/* ----- */

file:      main.h
project:   ModCon
developer: J.A. du Toit
company:   University of Stellenbosch

-----

```

```

2001/05/07 (ver.0.4.0):
Added enums for MCStage and MCState

2001/05/01 (ver.0.3.0):

*/

#ifndef _slave
#define _slave

/* ----- */

#include "MC_TYPES.h"
#include "MC_ERR.h"

/* ----- */

#define MC_PHASE_MAX      199
#define REF_COUNT_MAX     8

/* ----- */

#define LED_SEQ_START      0
#define LED_SEQ_DID        1
#define LED_SEQ_GID        2
#define LED_SEQ_MC_STATE   3
#define LED_SEQ_RES1       4
#define LED_SEQ_RES2       5
#define LED_SEQ_RES3       6
#define LED_SEQ_END        7

/* ----- */

#endif

/* -----< ModCon assembly functions >-----
-----

file:      F240_ASM.h
project:   PEG Code library
developer: J.A. du Toit
company:   University of Stellenbosch

-----

2001/05/08 (ver.0.4.0):
Replaced MC_Sin360 with MC_Sin(). Input is deg*10, output is scaled +/-32767

2001/05/01 (ver.0.3.0):

*/

#ifndef _F240_asm
#define _F240_asm

/* ----- */

char GetInfoChar(WORD Index);
WORD GetInfoLength(void);

/* ----- */

int MC_Sin(int);
/*
input is the angle in degrees*10 (0-3599)
returns an integer between -32767 and 32767
used for general fast sine calculations
No checking is done to verify the input angle
*/

int MC_Sin200(int);
/*
input is the angle in degrees (0-199)
returns an integer between -500 and 500
used primarily for PWM sine references
*/

int MC_IRef(int);
/*
input is the angle in degrees (0-199)
returns an integer between -500 and 500
*/

float MC_SinF200(int);
/*
input is the angle in degrees (0-199)
returns a float between -1 and 1
used primarily for PWM sine references
*/

/* ----- */

#endif

/* -----< ModCon Communications header file >-----
-----

file:      MC_COMS.h

```



```

project:    TMS320F240 library
developer:  J.A. du Toit
company:    University of Stellenbosch

-----

2001/05/01 (ver.0.3.0):

*/

#ifndef _MC_COMS
#define _MC_COMS

/* ----- */

#ifdef _MC_SLAVE
#include "MC_TYPES.h"
#endif

#ifdef _MC_MASTER
#include "MC_TYPES.h"
#endif

/* ----- */

/* define standard control characters */
typedef enum
{
    chNUL = 0x00,
    chDLE = 0x10,
    chSTX = 0x02,
    chETX = 0x03,
    chACK = 0x06,
    chCR  = 0x0D,
    chNAK = 0x15,
    chSYN = 0x16,
    chESC = 0x1B,
    chBOF = 0xC0,
    chEOF = 0xC1,
    chCE  = 0x7D
} chRS232;

/* ----- */

#define SCI_CMD_MASK      31
#define SCI_LENGTH_MASK  (3<<5)
#define SCI_RW_MASK      (1<<7)

#define SCI_GET_CMD_FIELD(x)      (( (x) & SCI_CMD_MASK ) >> 0)
#define SCI_GET_LENGTH_FIELD(x)  (( (x) & SCI_LENGTH_MASK) >> 5)
#define SCI_GET_RW_FIELD(x)      (( (x) & SCI_RW_MASK)   >> 7)

#define CMD(x, y, z)      (( (x) << 7) | ( (y) << 5) | ( (z) << 0) )

/* define host commands */
typedef enum
{
/* Command          RetPkt, Length, CMD_ID                                     */
CMD_NULL            = CMD(0, 0, 0x00), /* NULL command for testing purposes () */
CMD_SHUTDOWN        = CMD(0, 0, 0x01), /* shutdown target system ()            */
CMD_START           = CMD(0, 0, 0x02), /* start target ()                      */
CMD_STOP            = CMD(0, 0, 0x03), /* stop target ()                      */
CMD_SLEEP           = CMD(0, 0, 0x04), /* target enter sleep mode ()          */
CMD_WAKE            = CMD(0, 0, 0x05), /* wake target after sleep ()          */
CMD_SYNC            = CMD(0, 0, 0x06), /* synchronize target ()               */

CMD_RD_STATE        = CMD(1, 0, 0x01), /* read target state ()                */
CMD_RD_STAGE        = CMD(1, 0, 0x02), /* read target stage ()                */

CMD_RD_WDS          = CMD(1, 1, 0x10), /* read word from data space (addr A)  */
CMD_RD_WPS          = CMD(1, 1, 0x11), /* read word from program space (addr A) */
CMD_RD_WIO          = CMD(1, 1, 0x12), /* read word from I/O space (addr A)   */

CMD_WR_WDS          = CMD(0, 2, 0x10), /* write word to data space (addr A, data D) */
CMD_WR_WPS          = CMD(0, 2, 0x11), /* write word to program space (addr A, data D) */
CMD_WR_WIO          = CMD(0, 2, 0x12), /* write word to I/O space (addr A, data D) */

CMD_RD_INFO_CNT     = CMD(1, 0, 0x03), /* get number of info lines ()         */
CMD_RD_INFO_STR     = CMD(1, 1, 0x03), /* read info string (str n)           */

CMD_RD_WADS         = CMD(1, 2, 0x13), /* read word array from data space (addr A, length n) */
CMD_WR_WADS         = CMD(0, 3, 0x13), /* write word array to data space (length n, addr A, data[]) */

CMD_RD_REF          = CMD(1, 1, 0x18), /* read reference (ref n)              */
CMD_WR_REF          = CMD(0, 2, 0x18), /* write reference (ref n, data D)     */
CMD_WR_REF_BNDS     = CMD(0, 2, 0x19), /* write reference with bounds check (ref n, data D) */
CMD_RD_MEAS         = CMD(1, 1, 0x19), /* read measurement (meas n)          */

CMD_DACQ_SINGLE     = CMD(0, 0, 0x1A), /* single data acquisition              */
CMD_DACQ_SELECT     = CMD(0, 3, 0x1A), /* select number and sources for capture (Data[]) */

CMD_SET_TIME        = CMD(0, 2, 0x07), /* set timer tick count (timer L, timer H) */

CMD_CALC_CRC        = CMD(1, 2, 0x03) /* calc and return CRC (addr A, length n) */

} TComsCmd;

#undef CMD

/* ----- */

```

```

#ifdef _PEC_WORKSHOP

typedef struct
{
    int Cmd;
    char Name[ 20 ];
    char Description[ 40 ];
    char Parameters[ 20 ];
    WORD Flags;
} TCmdInfo;

#define MODCON_CMD_INFO_NUMBER    27
#define CMD_INFO_AVAIL            0x01

const TCmdInfo ModConCmdInfo[ MODCON_CMD_INFO_NUMBER ] =
{
    { CMD_NULL, "CMD_NULL", "NULL", "", CMD_INFO_AVAIL },
    { CMD_SHUTDOWN, "CMD_SHUTDOWN", "Shutdown device", "", CMD_INFO_AVAIL },
    { CMD_START, "CMD_START", "Start operation", "", CMD_INFO_AVAIL },
    { CMD_STOP, "CMD_STOP", "Stop operation", "", CMD_INFO_AVAIL },
    { CMD_SLEEP, "CMD_SLEEP", "Enter sleep mode", "", 0 },
    { CMD_WAKE, "CMD_WAKE", "Wake from sleep", "", 0 },
    { CMD_SYNC, "CMD_SYNC", "Synchronize timing", "", CMD_INFO_AVAIL },
    { CMD_RD_STATE, "CMD_RD_STATE", "Read target state", "", CMD_INFO_AVAIL },
    { CMD_RD_STAGE, "CMD_RD_STAGE", "Read target stage", "", CMD_INFO_AVAIL },
    { CMD_RD_WDS, "CMD_RD_WDS", "Read word from data space", "addr", CMD_INFO_AVAIL },
    { CMD_RD_WPS, "CMD_RD_WPS", "Read word from program space", "addr", 0 },
    { CMD_RD_WIO, "CMD_RD_WIO", "Read word from IO space", "addr", 0 },
    { CMD_WR_WDS, "CMD_WR_WDS", "Write word to data space", "addr, data", CMD_INFO_AVAIL },
    { CMD_WR_WPS, "CMD_WR_WPS", "Write word to program space", "addr, data", 0 },
    { CMD_WR_WIO, "CMD_WR_WIO", "Write word to IO space", "addr, data", 0 },
    { CMD_RD_INFO_CNT, "CMD_RD_INFO_CNT", "Read number of info lines", "", CMD_INFO_AVAIL },
    { CMD_RD_INFO_STR, "CMD_RD_INFO_STR", "Read info line", "num", CMD_INFO_AVAIL },
    { CMD_RD_WADS, "CMD_RD_WADS", "Read word array from data space", "addr, num", CMD_INFO_AVAIL },
    { CMD_WR_WADS, "CMD_WR_WADS", "Write word array to data space", "num, addr, data[]", 0 },
    { CMD_RD_REF, "CMD_RD_REF", "Read reference", "num", CMD_INFO_AVAIL },
    { CMD_WR_REF, "CMD_WR_REF", "Write reference", "num, data", CMD_INFO_AVAIL },
    { CMD_WR_REF_BNDS, "CMD_WR_REF_BNDS", "Write reference with bound check", "num, data", CMD_INFO_AVAIL },
    { CMD_RD_MEAS, "CMD_RD_MEAS", "Read measurement", "num", 0 },
    { CMD_DACQ_SINGLE, "CMD_DACQ_SINGLE", "Perform single data acquisition", "", CMD_INFO_AVAIL },
    { CMD_DACQ_SELECT, "CMD_DACQ_SELECT", "Select channels for acquisition", "data[]", CMD_INFO_AVAIL },
    { CMD_SET_TIME, "CMD_SET_TIME", "Set target time", "data_l, data_h", 0 },
    { CMD_CALC_CRC, "CMD_CALC_CRC", "Calculate CRC of block", "addr, num", CMD_INFO_AVAIL }
};

#ifdef

/* ----- */

/*
    Define return packet data types
    This 3 bit code is combined with the length parameter (5 bits) to form the
    packet info field. The length parameter includes only the data field.
*/

#define DESC_DATA_LENGTH_MASK    (0x1F)
#define DESC_DATA_TYPE_MASK     (0xE0)

#define DATA_TYPE_CONTROL      (0<<5)    /* control codes */
#define DATA_TYPE_INT_ARRAY    (1<<5)    /* int array */
#define DATA_TYPE_UINT_ARRAY   (2<<5)    /* unsigned int array */
#define DATA_TYPE_CHAR_ARRAY   (3<<5)    /* character array */
#define DATA_TYPE_FLOAT_ARRAY  (4<<5)    /* float array */
#define DATA_TYPE_MEAS_ARRAY   (5<<5)    /* measurements (reserve) */
#define DATA_TYPE_RES0         (6<<5)    /* reserve */
#define DATA_TYPE_REJECT       (7<<5)    /* reject command */

/* ----- */

/* define target to host packet indexes */
#define TH_PKT_HID_INDEX        0
#define TH_PKT_TID_INDEX        (TH_PKT_HID_INDEX + 1)
#define TH_PKT_DESC_INDEX       (TH_PKT_TID_INDEX + 1)
#define TH_PKT_DATA0_INDEX      (TH_PKT_DESC_INDEX + 1)
#define TH_PKT_NUM_OF_HEADER_BYTES  3
#define TH_PKT_NUM_CONTROL_BYTES  (TH_PKT_NUM_OF_HEADER_BYTES + 1)

/* ----- */

#endif

/*
    ---< ModCon SCI library header file >---
    -----

    file:      MC_DACQ.h
    project:    Data acquisition for ModCon
    developer:  J.A. du Toit
    company:    University of Stellenbosch
    -----

    2001/11/26 (ver.0.1.0)

    */

#ifdef _MC_DACQ
#define _MC_DACQ

/* ----- */

```



```

#include "MC_TYPES.h"

/* ----- */

#define DACQ_START      (1<<0)
#define DACQ_START_TRIG (1<<1)
#define DACQ_BUSY      (1<<2)

#define DACQ_SRC_MAXNUM 16

#define DACQ_ADDR_START (int *)0xB000
#define DACQ_ADDR_END   (int *) (0xC000-DACQ_SRC_MAXNUM)

/* ----- */

extern int DacqFlags;
extern int DacqSrcCnt;
extern int *DacqDst;
extern int DacqSrc[];

/* ----- */

static inline void DoDACQ(void)
{
    int i;

    /* check if busy */
    if (DacqFlags & DACQ_BUSY)
    {
        for (i=0; i<DacqSrcCnt; i++) *DacqDst++ = *(int*)(DacqSrc[i]);
        if (DacqDst > DACQ_ADDR_END) DacqFlags &= ~DACQ_BUSY;
    }

    /* check if start flag is set */
    else if ((DacqFlags & (DACQ_START|DACQ_START_TRIG)) == (DACQ_START|DACQ_START_TRIG))
    {
        /* clear start flag and set busy flag */
        DacqFlags = DACQ_BUSY;

        /* point to start of DACQ storage space */
        DacqDst = DACQ_ADDR_START;
    }
}

/* ----- */

#endif

/* ----- ModCon EPLD header file ----- */

-----

file:      MC_EPLD.h
project:   TMS320F240 library
developer: J.A. du Toit
company:   University of Stellenbosch

-----

2001/04/24 (ver.0.3.0):
Modified led control flags to sync with epld version

*/

#ifndef _MC_EPLD
#define _MC_EPLD

/* ----- */

#include <ioports.h>

/* ----- */

typedef enum
{
    CB_SCI_BYPASS      = 1<<0,
    CB_REF_SYNC_ENA    = 1<<1,
    CB_CLR_PDPINT_SRC  = 1<<2,
    CB_SYNC_ENA        = 1<<3,
    CB_BREAK_ENA       = 1<<4,
    CB_IGBT_ERR_ENA    = 1<<5,
    CB_OC_ENA          = 1<<6
} TEpldControlBit;

typedef enum
{
    SB_LINE_BREAK      = 1<<0,
    SB_LEM_OC          = 1<<1,
    SB_ERR_TOP0        = 1<<2,
    SB_ERR_BOT0        = 1<<3,
    SB_ERR_TOP1        = 1<<4,
    SB_ERR_BOT1        = 1<<5
} TEpldStatusBit;

#ifdef _MC_MASTER
#define EPLD_CONTROL_WAIT_SYNC_CLR (CB_CLR_PDPINT_SRC|CB_SYNC_ENA)
#define EPLD_CONTROL_WAIT_SYNC_DONE (CB_CLR_PDPINT_SRC|CB_SYNC_ENA|CB_BREAK_ENA)
#define EPLD_CONTROL_RUN (CB_SYNC_ENA|CB_BREAK_ENA)
#define EPLD_CONTROL_SHUTDOWN (CB_SCI_BYPASS)
#endif

```

```

#ifndef _MC_SLAVE
#define EPLD_CONTROL_WAIT_SYNC_CLR (CB_CLR PDPINT_SRC)
#define EPLD_CONTROL_WAIT_SYNC_DONE (CB_CLR PDPINT_SRC|CB_SYNC_ENA|CB_BREAK_ENA|CB_OC_ENA|CB_IGBT_ERR_ENA)
#define EPLD_CONTROL_RUN (CB_SYNC_ENA|CB_BREAK_ENA|CB_OC_ENA|CB_IGBT_ERR_ENA)
#define EPLD_CONTROL_SHUTDOWN (CB_SCI_BYPASS)
#endif

/* ----- */

typedef enum
{
    EPLD_CONTROL = 0x0,
    EPLD_VERSION = 0x1,
    EPLD_TID = 0x2,
    EPLD_LEDS = 0x3,
    EPLD_LED0 = 0x4,
    EPLD_LED1 = 0x5,
    EPLD_LED2 = 0x6,
    EPLD_LED3 = 0x7,
    EPLD_DAC0 = 0x8,
    EPLD_DAC1 = 0x9,
    EPLD_DAC2 = 0xA,
    EPLD_DAC3 = 0xB
} TEpldRegister;

/* ----- */

typedef enum
{
    LED_FLASH = 0x0,
    LED_OFF = 0x1,
    LED_DIMM = 0x2,
    LED_ON = 0x3,
    LED_SCI = 0x4
} TLedState;

/* ----- */

static inline void EPLD_RegWrite(TEpldRegister addr, int data)
{
    outport(addr, data);
}

/* ----- */

static inline int EPLD_RegRead(TEpldRegister addr)
{
    /* read a value from the EPLD (byte) */

    int ret;
    inport(addr, &ret);
    return (ret & 0xFF);
}

/* ----- */

#endif

/* -----< PEG code library error definitions >-----
-----

file:      MC_ERR.h
project:   PEG code library
developer: J.A. du Toit
company:   University of Stellenbosch

-----

2001/05/01 (ver.0.1.0):

*/

#ifndef _MC_ERR
#define _MC_ERR

/* ----- */

typedef enum
{
    /* Misc */
    ERR_NONE = 0x10,
    ERR_HOST_REQ = 0x11,
    ERR_LIB_VER = 0x12,
    ERR_END_OF_CODE = 0x13,

    /* Timing */
    ERR_WDT_TRIP = 0x22,
    ERR_ILLL_ADDR = 0x23,
    ERR_SCI_NR_TIMEOUT = 0x24,

    /* Current */
    ERR_OC_AC_HARD = 0x31,
    ERR_OC_AC_SOFT = 0x32,

    /* Voltage */
    ERR_OV_DC = 0x41,

```



```

ERR_OV_AC          = 0x42,

/* Temperature */
ERR_OT_PCB         = 0x51,
ERR_OT_HS          = 0x52,

/* stage timing */
ERR_STAGE_SYNC     = 0x61,
ERR_INV_STAGE      = 0x62,
ERR_SYNC_LOST      = 0x63,

/* IGBT status */
ERR_IGBT_GROUP     = 0x70,
ERR_IGBT_TOP0      = 0x71,
ERR_IGBT_BOT0      = 0x72,
ERR_IGBT_TOP1      = 0x74,
ERR_IGBT_BOT1      = 0x78,

/* line status */
ERR_BREAK_DET      = 0x81,
ERR_INV_PDPINT_SRC = 0x82,
ERR_SCI_BREAK_DET  = 0x83,

/* power supply */
ERR_OC_PSUP        = 0x91,
ERR_OV_PSUP        = 0x92,

/* undefined */
ERR_UNDEF          = 0xFF

} TF240Error;

/* ----- */

#endif

/* -----< ModCon library header file >-----
-----

file:      MC_LIB.h
project:   TMS320F240 library
developer: J.A. du Toit
company:   University of Stellenbosch

-----
2001/07/12 (ver.0.5.0):
Added valid address checking function for data space

2001/05/01 (ver.0.3.2):

*/

#ifndef MC_LIB
#define MC_LIB

/* ----- */

#include "MC_TYPES.h"
#include "MC_ERR.h"

/* ----- */

#define SYSCLK_FREQ ((long)10000000)
#define ADC_MAX_ENTRIES 8

/* ----- */

typedef enum
{
    OPSTATE_UNDEF,
    OPSTATE_NORM_RUN,
    OPSTATE_NORM_IDLE,
    OPSTATE_TIMEOUT_IDLE,
    OPSTATE_SYNC_LOST,
    OPSTATE_SHUTDOWN,
    OPSTATE_SLEEP,
    OPSTATE_TIFLSH,
    OPSTATE_WAIT_SYNC_LOCK,
    OPSTATE_WAIT_STAGE_SYNC
} TopState;

/* ----- */

typedef enum
{
    OPSTAGE_STAGE0 = 0,
    OPSTAGE_STAGE1 = 1,
    OPSTAGE_STAGE2 = 2,
    OPSTAGE_STAGE3 = 3,
    OPSTAGE_SYNC_LOCK = 4,
    OPSTAGE_STAGE_SYNC = 5
} TopStage;

/* ----- */

#define INFO_COUNT_MAX 62
#define INFO_SLENGTH_MAX 32
#define INFO_START_ADDR (WORD)0x8000
#define INFO_LENGTH_ADDR INFO_START_ADDR

```

```

#define INFO_COUNT_ADDR      INFO_LENGTH_ADDR+1
#define INFO_INDEX_ADDR     INFO_COUNT_ADDR+1
#define INFO_TABLE_ADDR     INFO_INDEX_ADDR + 2*INFO_COUNT_MAX

#define T1PERIOD_NOM        1000
#define T1PERIOD_MAX        1001
#define T1PERIOD_MIN        999

#define SYNC_LOCK_TIMEOUT   10

/* ----- */

void MC_Init(void);
void MC_Shutdown(int InitError);

void PWM_Init(void);
void CAP_Init(void);

void EPLD_LEDSpin(int Speed);

/* ----- */

static inline int IsValidDSAddr(void *addr)
{
    /*
     * check if given address is a valid F240 data space address
     * invalid blocks are :
     * [1] 0x0800 to 0x6FFF
     * [2] 0x7800 to 0x7FFF
     * Any access to the above blocks will cause an invalid address reset state
     */

    if (addr < (void*)0x0800) return TRUE;
    else if (addr >= (void*)0x8000) return TRUE;
    else if ((addr >= (void*)0x7000) && (addr < (void*)0x7800)) return TRUE;
    else return FALSE;
}

/* ----- */

#endif

/*
 * -----< ModCon PWM reference header file >-----
 * -----
 *
 * file:      MC_PWM.h
 * project:   Data acquisition for ModCon
 * developer: J.A. du Toit
 * company:   University of Stellenbosch
 *
 * -----
 *
 * 2001/11/29 (ver.0.1.0)
 *
 *
 * #ifndef MC_PWM
 * #define MC_PWM
 *
 * /* ----- */
 *
 * #include "MC_Types.h"
 *
 * /* ----- */
 *
 * typedef enum
 * {
 *     PWM_REF_50,
 *     PWM_REF_SIN_OL,
 *     PWM_REF_SIN_IREG,
 *     PWM_REF_SINSQR_OL,
 *     PWM_REF_SINSQR_IREG
 * } TRefType;
 *
 * /* ----- */
 *
 * extern int MeasVdcb[];
 * extern int MeasVout[];
 * extern int MeasIout[];
 * extern int Ref[];
 *
 * extern int MCPhase;
 *
 * /* ----- */
 *
 * static inline void DoPWM(void)
 * {
 *     int IRef;
 *     int Sine;
 *     int PWM0dc;
 *     int PWM1dc;
 *     int PWMRef;
 *
 *     switch (Ref[3])
 *     {
 *         case PWM_REF_SIN_OL:
 *             {

```



```

        IRef = MC_Sin200(MCPhase);
        IRef = ((long)IRef * (long)Ref[0]) >> 16;
        PWMRef = IRef;
        break;
    }

    case PWM_REF_SIN_IREF:
    {
        IRef = MC_Sin200(MCPhase);
        IRef = ((long)IRef * (long)Ref[0]) >> 16;
        PWMRef = Ref[1] * (IRef - MeasIout[1] + 0x200);
        break;
    }

    case PWM_REF_SINSQR_OL:
    {
        IRef = MC_IRef(MCPhase);
        IRef = ((long)IRef * (long)Ref[0]) >> 16;
        PWMRef = IRef;
        break;
    }

    case PWM_REF_SINSQR_IREF:
    {
        IRef = MC_IRef(MCPhase);
        IRef = ((long)IRef * (long)Ref[0]) >> 16;
        PWMRef = Ref[1] * (IRef - MeasIout[1] + 0x200);
        break;
    }

    default:
    {
        IRef = 0;
        PWMRef = 0;
        break;
    }
}

/* check for overflow */
if (PWMRef > 500) PWMRef = 500;
if (PWMRef < -500) PWMRef = -500;

/* distribute zero states */
PWM0dc = 500 - PWMRef;
PWM1dc = 500 + PWMRef;

/* write duty cycles to double buffers */
iaCMPR2 = PWM0dc;
iaCMPR1 = PWM1dc;
}

/* ----- */
#endif

/* -----< ModCon SCI library header file >-----
-----
file:      MC_SCI.h
project:   TMS320F240 library
developer: J.A. du Toit
company:   University of Stellenbosch
-----

2001/07/03 (ver.0.5.0)
2001/05/01 (ver.0.3.0)
*/

#ifndef _MC_SCI
#define _MC_SCI

/* ----- */

#include "MC_TYPES.h"
#include "MC_COMS.h"
#include "F240REGS.h"
#include "MC_LIB.h"

/* ----- */

#define TI_FLASH_COUNT_MAX      256

#define SCI_NUM_OF_DATA_BYTES    32      /* 32 bytes of data */
#define SCI_NUM_OF_CONTROL_BYTES 4       /* CRC, TID, CMD or CRC, TID HID, DESC */

#define SCI_BUFF_LENGTH (SCI_NUM_OF_CONTROL_BYTES + SCI_NUM_OF_DATA_BYTES)

#define SCI_WDT_COUNT_MAX      25000    /* 25000 * 200us steps => 5s */

/* SCI receiver circular buffer */
#define SCI_RXBUFF_MASK        0x0F      /* 2^n-1 */
#define SCI_RXBUFF_LENGTH      (SCI_RXBUFF_MASK+1) /* must be 2^n, n>2 */

#define HID                     0x7F     /* Host ID (recipient of transmitted packets) */

/* ----- */

```

```

/* define extended packet lengths */
#define SCI_CMD_LENGTH_0      3
#define SCI_CMD_DWNUM_0      0

#define SCI_CMD_LENGTH_1      5
#define SCI_CMD_DWNUM_1      1

#define SCI_CMD_LENGTH_2      7
#define SCI_CMD_DWNUM_2      2

#define SCI_CMD_LENGTH_3      4

/* define receive buffer indexes */
#define SCI_BUFF_TID_INDEX    0
#define SCI_BUFF_CMD_INDEX    1
#define SCI_BUFF_DATA0_INDEX  2
#define SCI_BUFF_DATA1_INDEX  4
#define SCI_BUFF_DLEN_INDEX   2

/* ----- */

/* SCI state machines */
typedef enum
{
    SCI_TX_IDLE,
    SCI_TX_DBUFF,
    SCI_TX_BUSY,
    SCI_TX_BUSY_CE,
    SCI_TX_BLOCK,
    SCI_TX_BOF,
    SCI_TX_EOF
} TsciTxState;

typedef enum
{
    SCI_IDLE,
    SCI_BUSY,
    SCI_BUSY_CE,
    SCI_EOF,
    SCI_SLEEP,
    SCI_BYPASS,
    SCI_BLOCK
} TsciState;

/* ----- */

#define TARGET_NO      0x00
#define TARGET_ONLY    0x01
#define TARGET_GROUP   0x02
#define TARGET_ALL     0x04

/* ----- */

void SCI_Init(TbaudRate baud);

inline int SCI_CheckTID(int ID);

int SCI_SendInfoLine(WORD Line);

int SCI_SendWA(int *data, int num, int sign);
int SCI_SendCA(char *data, WORD num);

void SCI_ProcessRxBytes(int Num);
int SCI_ProcessPkt(void);

int SCI_ProcessCmd0(TComsCmd CMD);
int SCI_ProcessCmd1(TComsCmd CMD, WORD data);
int SCI_ProcessCmd2(TComsCmd CMD, WORD data0, WORD data1);
int SCI_ProcessCmd3(TComsCmd CMD, BYTE DLength, char *data);

/* ----- */

static inline void SCI_AddRxChar(char data)
{
    extern WORD iRxIn;
    extern WORD iRxOut;
    extern char RxBuff[];
    extern TSCIErrors SCIErrors;
    extern TsciState SCIState;

    /* add character to SCI Rx buffer, checking for overflow */
    if ((iRxIn-iRxOut+1) & SCI_RXBUFF_MASK)
    {
        iRxIn = (iRxIn+1) & SCI_RXBUFF_MASK;
        RxBuff[iRxIn] = data;
    }
    else
    {
        /* Log error if SCIState not sciEOF */
        if (SCIState != SCI_EOF) SCIErrors.RxOF++;
    }
}

/* ----- */

static inline int SCI_AddChar(char data)
{
    extern char SCIBuff[];
    extern WORD iBuffIn;

```



```

extern int SCI_CRC;

/* add character to SCI buffer, checking for overflow */
if (iBuffIn < SCI_BUFF_LENGTH)
{
    SCIBuff[iBuffIn] = data;
    iBuffIn++;
    SCI_CRC += data;
    return TRUE;
}
else
{
    return FALSE;
}
}

/* ----- */

static inline char SCI_GetRxChar(void)
{
    /* get character from Rx buffer */
    char data;
    extern WORD iRxIn;
    extern WORD iRxOut;
    extern TSCIErrors SCIErrors;
    extern char RxBuff[];

    /* check if data available, else return chBOF */
    if (iRxIn != iRxOut)
    {
        iRxOut = (iRxOut+1) & SCI_RXBUFF_MASK;
        data = RxBuff[iRxOut];
    }
    else
    {
        data = chBOF;
        /* log error */
        SCIErrors.RxUR++;
    }

    return data;
}

/* ----- */

static inline int SCI_GetRxNum(void)
{
    extern WORD iRxIn;
    extern WORD iRxOut;

    /* read number of new chars in SCI RX buffer */
    int Num;
    /*asm (" CLRC INTM");*/
    Num = (iRxIn-iRxOut) & SCI_RXBUFF_MASK;
    /*asm (" SETC INTM");*/
    return Num;
}

/* ----- */

static inline void SCI_SendChar(char data)
{
    extern TsciTxState SCITxState;
    extern char SCITxDBuff;

    /* write to SCITXBUF if empty, else write to dbuff and signal */
    if (SCITxState == SCI_TX_IDLE)
    {
        /* check if SCITXBUF is empty */
        if (iaSCICTL2 & TXRDY)
        {
            iaSCITXBUF = data;
        }
        else
        {
            /* store in double buffer and signal */
            SCITxDBuff = data;
            SCITxState = SCI_TX_DBUFF;
        }
    }
}

/* ----- */

static inline void SCI_DoRx(void)
{
    WORD tmpSCIRXST = iaSCIRXST;
    extern WORD SCIWDTCnt;
    extern TSCIErrors SCIErrors;

    char SCIRxD;

    tmpSCIRXST = iaSCIRXST;

    /* check for receive errors */
    if (tmpSCIRXST & RXERROR)
    {
        #ifdef MC_MASTER
            /* Check for break detect => shutdown */

```

```

    if (tmpSCIRXST & RXERROR_BRKDT) MC_Shutdown(ERR_SCI_BREAK_DET);
#endif

    /* reset SCI (SCI Software Reset: hi, low, hi) */
    iaSCICTL1 = defSCICTL1r;
    iaSCICTL1 = defSCICTL1;
    SCIErrors.SCI++;

}

/* check for new data */
else if (tmpSCIRXST & RXRDY)
{
    /* new data in input SCIRX buffer, clear SCI watchdog timer */
    SCIWDTcnt = 0;

    /* read, store and send the SCI data */
    SCIRxD = iaSCIRXBUF & 0xFF;
    SCI_SendChar(SCIRxD);
    SCI_AddRxChar(SCIRxD);
}

}

/* ----- */
static inline void SCI_DoTx(void)
{
    extern TsciTxState SCITxState;
    extern TsciState SCIState;
    extern char SCITxDBuff;
    extern char SCIBuff[];
    extern WORD iBuffOut;
    extern WORD nBuffOut;

    char SCITxD;

    /* check if the transmit buffer is ready to receive new data */
    if (iaSCICTL2 & TXRDY)
    {
        switch (SCITxState)
        {
            case SCI_TX_IDLE:
            {
                break;
            }

            case SCI_TX_DBUFF:
            {
                /* send data in Tx double buffer */
                iaSCITXBUF = SCITxDBuff;
                SCITxState = SCI_TX_IDLE;
                break;
            }

            case SCI_TX_BOF:
            {
                /* send BOF character and reset Buff index */
                iaSCITXBUF = chBOF;
                iBuffOut = 0;
                SCITxState = SCI_TX_BUSY;
                break;
            }

            case SCI_TX_EOF:
            {
                iaSCITXBUF = chEOF;
                SCITxState = SCI_TX_IDLE;
                SCIState = SCI_IDLE;
                break;
            }

            case SCI_TX_BUSY:
            {
                SCITxD = SCIBuff[iBuffOut] & 0xFF;

                /* check for control characters, CE, BOF and EOF */
                if ((SCITxD == chCE) || (SCITxD == chBOF) || (SCITxD == chEOF))
                {
                    iaSCITXBUF = chCE;
                    SCITxState = SCI_TX_BUSY_CE;
                }
                else
                {
                    /* transmit character and check if end of frame reached */
                    iaSCITXBUF = SCITxD;
                    if (++iBuffOut == nBuffOut)
                    {
                        SCITxState = SCI_TX_EOF;
                    }
                }
                break;
            }

            case SCI_TX_BUSY_CE:
            {
                /* transmit character xor 0x20 */
                SCITxD = SCIBuff[iBuffOut] & 0xFF;
                iaSCITXBUF = SCITxD^0x20;
            }
        }
    }
}

```



```

        /* check if end of frame reached */
        if (++iBuffOut == nBuffOut)
        {
            SCITxState = SCI_TX_EOF;
        }
        else
        {
            SCITxState = SCI_TX_BUSY;
        }
        break;
    }
}

default:
{
    break;
}
}
}

/* ----- */

static inline void SCI_DoComs(void)
{
    int Num;
    int i;
    extern TsciState SCIState;
    extern TsciTxState SCITxState;

    /* check for new data in Rx buffer if transmitter state is Idle (unified Tx/Rx buffer) */
    if ((SCITxState == SCI_TX_IDLE) || (SCITxState == SCI_TX_DBUFF))
    {
        /* get number of unread characters in Rx buffer */
        Num = SCI_GetRxNum();

        if (Num>0)
        {
            /* process new data */
            SCI_ProcessRxBytes(Num);

            /* check if EOF received */
            if (SCIState == SCI_EOF)
            {
                SCI_ProcessPkt();
            }
        }
    }
}

/* ----- */

static inline int CalcCRC(char *data, WORD num)
{
    /* this iteration calculates the checksum of data pointed to by data, length num */
    int i;
    int CRC = 0;

    for (i=0; i<num; i++) CRC += data[i];

    return CRC;
}

/* ----- */

static inline void SCI_BuffReset(void)
{
    extern WORD iBuffIn;
    extern WORD iBuffOut;

    extern WORD iRxIn;
    extern WORD iRxOut;

    extern TsciState SCIState;
    extern TsciTxState SCITxState;

    asm (" SETC INTM");

    iBuffIn = 0;
    iBuffOut = 0;

    iRxIn = 0;
    iRxOut = 0;

    /* reset states and initialize all buffers */
    SCIState = SCI_IDLE;
    SCITxState = SCI_TX_IDLE;

    asm (" CLRC INTM");
}

/* ----- */

#endif

/* -----> ModCon Typaes file >----- */

```

```

-----
file:      MC_TYPES.h
project:   TMS320F240 library
developer: J.A. du Toit
company:   University of Stellenbosch
-----

2001/05/01 (ver.0.2.0):

*/

#ifndef _MC_TYPES
#define _MC_TYPES

/* ----- */

typedef int BYTE;
typedef unsigned long ULONG;
typedef unsigned int WORD;
typedef volatile unsigned int VWORD;

#define TRUE 1
#define FALSE 0

#define pi 3.14159265

/* ----- */

/* struct that holds the runtime data for the extended integer-info package */

typedef struct
{
    int *addr;      /* address of data source */
    int data;       /* last values stored */
    int max;        /* maximum value stored */
    int min;        /* minimum values stored */
    long acc;       /* accumulated value */
    WORD cnt;       /* number of samples accumulated */
    WORD cntmax;    /* max number of accumulated values */
    long timestamp; /* timestamp of last value */
    int control;    /* control flags */
    int status;     /* status flags */
    int CueID;      /* Packet ID for cued transmission */
} TIntDataEx;

/* ----- */

/* struct that holds the runtime data for the extended unsigned integer-info package */

typedef struct
{
    int *addr;      /* address of data source */
    WORD data;      /* last values stored */
    WORD max;       /* maximum value stored */
    WORD min;       /* minimum values stored */
    long acc;       /* accumulated value */
    WORD cnt;       /* number of samples accumulated */
    WORD cntmax;    /* max number of accumulated values */
    long timestamp; /* timestamp of last value */
    int control;    /* control flags */
    int status;     /* status flags */
    int CueID;      /* Packet ID for cued transmission */
} TUIntDataEx;

/* ----- */

#endif

; -----
; -----< MC_asm assembly functions >-----
; -----
;
; file:      MC_asm.asm
; project:   Module Controller (modcon)
; developer: J.A. du Toit
; company:   University of Stellenbosch
; -----

; 2001/05/08 (ver.0.5.0)
; Added MC_Sin routine

; -----

chNUL .set 0
chLF .set 10
chCR .set 13

; reserve, add later

; function int = MC_Sin
; input is the angle in degrees (0-359) in steps of 0.1deg
; returns an integer between -32767 and 32767
; used for general fast sine calculations

```



page 190

```

; function char = GetInfo(WORD Index)

.global _GetInfoChar
_GetInfoChar:
    POPD    ++
    SAR     AR0, ++
    SAR     AR1, *
    SBRK    3

    RSXM
    LALK    Info_Start
    ADD     *
    TBLR    *
    LACL    *

    ADRK    3
    MAR     *, AR1
    SBRK    1
    LAR     AR0, *-
    PSHD    *
    RET

; -----

; function WORD = GetInfoLength(void)

.global _GetInfoLength
_GetInfoLength:
    RSXM
    LALK    Info_End
    SBLK    Info_Start
    RET

; -----

Info_Start:
    .byte "%LibName ModCon.lib",      chNUL
    .byte "%LibVersion 050",          chNUL
    .byte "%ProgName ModCon",         chNUL
    .byte "%ProgVersion ModCon",      chNUL
    .byte "%DID 0",                   chNUL
    .byte "%GID 0",                   chNUL
    .byte "%Meas0 Vdcb",               chNUL
    .byte "%Meas1 Vout",               chNUL
    .byte "%Meas2 Iout",               chNUL
    .byte "%Meas3 none",               chNUL
    .byte "%Meas4 none",               chNUL
    .byte "%Meas5 none",               chNUL
    .byte "%Meas6 none",               chNUL
    .byte "%Meas7 none",               chNUL
    .byte "%Meas8 none",               chNUL
    .byte "%Meas9 none",               chNUL
    .byte "%Meas10 none",              chNUL
    .byte "%Meas11 none",              chNUL
    .byte "%Meas12 none",              chNUL
    .byte "%Meas13 none",              chNUL
    .byte "%Meas14 none",              chNUL
    .byte "%Meas15 none",              chNUL
    .byte "%Ref0 Reference scale",     chNUL
    .byte "%Ref1 Iref proportional gain", chNUL
    .byte "%Ref2 Iref integration const", chNUL
    .byte "%Ref3 Reference type",      chNUL
    .byte "%Ref4 none",                chNUL
    .byte "%Ref5 none",                chNUL
    .byte "%Ref6 none",                chNUL
    .byte "%Ref7 none",                chNUL

Info_End:

; -----
; reserve, add later

MC_Sin_Start

; -----

MC_Sin200_Start:
    .int    0
    .int    16
    .int    -31

; -----

MC_SinF200_Start:
    .float  0.0000000
; .float  -0.0627905

; -----

MC_IRef_Start:
    .int    0
    .int    -31

```



```

; -----
;
; .end
;
; -----< F240 Reset vector >-----
;
; -----
;
; file:      F240_res.asm
; project:   Module Controller (modcon)
; developer: J.A. du Toit
; company:   University of Stellenbosch
; date:      08/01/2001
; version:   0.9.0
;
; -----
;
; update 20/02/2001   added wait state generator initialization code
;
; .global _c_int0 ; c environment entry point
;
; define addresses
DP_FF1 .set 224      ; page 1 of peripheral file (7000h/80h)
SYSSR .set 0701Ah    ; System Module Status Register
RTICR .set 07027h    ; RTI Control register
WDCR .set 07029h     ; WD Control register
CKCR0 .set 0702Bh    ; PLL Clock Control Register 0
CKCR1 .set 0702Dh    ; PLL Clock Control Register 1
WSGR .set 0FFFFh     ; Wait State Generator register (in I/O space)
DP_B0 .set 4         ; DARAM_B0 data page
;
TBITS .set 000Ah
TBITS15 .set 0000h
;
; code starts here (map to 40h for TI SCI bootloader)
;
CLRC   CNF      ; see Memory map. CNF=0 @ reset
SETC   INTM     ;
CLRC   SXM      ;
CLRC   XF       ;
SETC   INTM     ; disable interrupts
;
; set WSGR to 0 => address visibility and wait states to 0
LDP    #DP_B0
SPLK   #0, 0    ; 0 => no wait states and avis=0
OUT    0, WSGR
;
; Set DP to first peripheral file page, 0x7000-7f.
LDP    #DP_FF1
;
; initialize WDT registers
SPLK   #06Fh, WDCR ; clear WDFLAG, Disable WDT, set WDT for 1 second overflow (max)
SPLK   #07h, RTICR ; clear RTI Flag, set RTI for 1 second overflow (max)
;
; detect reset source
PORCHK: BIT    SYSSR, TBITS15
BCND   NOTPOR, NTC
;
; setup PLL, div/2, CPUCLK=20MHz, SYSCLK=10MHz
SPLK   #0001h, CKCR0
SPLK   #0068h, CKCR1
SPLK   #0081h, CKCR0
;
LACC   SYSSR     ; clear status: POR, ILLADR WDTRST, SWRST
AND    #0FFh
SACL   SYSSR
;
; wait for PLL to lock
PLLCHK: BIT    CKCR0, TBITS15
BCND   PLLCHK, NTC
B      END
;
NOTPOR:
SPLK   #0081h, CKCR0 ; set SYSCLK to CPUCLK/2
;
END:   B      _c_int0 ; jump to boot.asm for c environment initialization
; -----

```

# Appendix D

---

---

PC software source



## D. PC software source

The code listed in the appendix is an excerpt from the client and server applications.

For brevity, only the header files of the relevant classes are shown.

### Server code

```

typedef struct
{
    DWORD SvrCmd;           // command to server (send packet, server control etc.)
    DWORD PktID;            // packet ID
    DWORD Timeout;          // packet timeout on server
    DWORD Flags;
} TNPipeCnInfo;

typedef enum
{
    SEND_TARGET_PACKET,
    NPIPE_REQ_CLOSE,
    NPIPE_READ_STATUS,
    LINK_READ_STATUS,
    SERVER_READ_STATUS,
    LINK_DISCONNECT,
    LINK_CONNECT,
    LINK_WRITE_SETTINGS,
    LINK_READ_SETTINGS
} TNPipeCmd;

typedef struct
{
    DWORD DataAvailable;
    DWORD PipeConnected;
} TNPipeStatus;

//-----

class TNPipeServiceThreadObj : public TThread
{
private:
protected:
    // misc
    TPipe **Pipes;
    BYTE PipeCount;
    BYTE ActivePipe;

    HANDLE *ObjHandles;
    DWORD HandleCount;

    AnsiString PipeName;
    TNPipeStatus PipeStatus;

    virtual void __fastcall Execute() = 0;
    virtual void DebugMessage(AnsiString Message) = 0;
    virtual bool SendTargetFrame(BYTE HandleIdx) = 0;
    virtual bool SendPipeMessage(BYTE HandleIdx, DWORD Flags, BYTE *Pkt, DWORD Num) = 0;
    virtual bool ReadPipeMessage(DWORD PipeNum) = 0;

public:
    TStatusInfo **StatusInfo;
    DWORD StatusInfoCount;

    TMessageLog *MessageLog;

    __fastcall TNPipeServiceThreadObj(bool CreateSuspended, DWORD APipeCount);
    __fastcall virtual ~TNPipeServiceThreadObj(void);
};

//-----

#endif

//-----

#ifndef ModCon232ServiceThreadH
#define ModCon232ServiceThreadH

//-----

#include <Classes.hpp>

#pragma hdrstop

#include "NPipeServiceThreadObj.h"
#include "Comport.h"

```

```

#include "Pipe.h"
#include "EventTimer.h"
#include "ModCon.h"
#include "ModCon232Status.h"
// #include "FIFO.h"

//-----

#define OBJIDX_WAIT_TIMER      0
#define OBJIDX_COM_EVENT      OBJIDX_WAIT_TIMER + 1
#define OBJIDX_COM_READ       OBJIDX_COM_EVENT + 1
#define OBJIDX_COM_WRITE      OBJIDX_COM_READ + 1
#define OBJIDX_PIPE_START     OBJIDX_COM_WRITE + 1

#define MAX_PACKET_SIZE       32 + 2

#define RETRY_CNT_MAX         3

//-----

class TModCon232ServiceThread : public TNPipeServiceThreadObj
{
private:
    BYTE HID;

    int RetryCnt;

    // misc
    TComPort *ComPort;
    TEventTimer *Timer;

    DWORD PipeBufferSize;
    BYTE *PktBuff;
    BYTE MaxPktSize;

    int ComPortNum;

    bool ReadPipeMessage(DWORD PipeNum);
    DWORD EncodeTargetFrame(BYTE *InBuffer, BYTE *OutBuffer,  DWORD Num);
    bool DecodeTargetFrame(BYTE* Buffer,  DWORD Num);
    bool CancelMessage(BYTE HandleIdx, bool SendNACK);

protected:
    void __fastcall Execute();

    virtual void DebugMessage(AnsiString Message);
    virtual bool SendTargetFrame(BYTE HandleIdx);
    virtual bool SendNextTargetFrame(void);
    virtual bool RetrySendTargetFrame(void);
    virtual bool SendPipeMessage(BYTE HandleIdx, DWORD Flags, BYTE *Pkt, DWORD Num);

public:
    __fastcall TModCon232ServiceThread(bool CreateSuspended, DWORD APipeCount);
    __fastcall virtual ~TModCon232ServiceThread(void);

    void UpdateStatus(void);

    TModCon232StatusFrame *StatusDisplay;
};

//-----

#endif

//-----

#ifndef ComPortH
#define ComPortH

//-----

#include <windows.h>

#pragma hdrstop

#include "..\ModCon.100\Shared\Source\MC_COMS.h"
#include "PWEError.h"

//-----

typedef struct
{
    AnsiString PortName;
    DWORD BaudRate;
} TComPortSettings;

class TComPort
{
private:
    DWORD ReadBufferSize;

```



```

    DWORD WriteBufferSize;

protected:

public:

    bool ComPortOpen;
    bool PendingRead;
    bool PendingWrite;
    bool PendingEvent;

    DWORD Status;
    DWORD EventMask;

    DWORD BytesToWrite;
    DWORD BytesRead;

    // read and write buffers
    BYTE  NumOfBuffers;
    BYTE **WriteBuffer;
    DWORD *NumOfBytes;
    BYTE  *ReadBuffer;
    DWORD *NumOfRetPackets;

    // handle to comport
    HANDLE Handle;

    TComPortSettings PortSettings;

    // event objects and overlapped structures used for overlapped I/O
    HANDLE EventEvHandle;
    HANDLE ReadEvHandle;
    HANDLE WriteEvHandle;

    OVERLAPPED EventOverlap;
    OVERLAPPED ReadOverlap;
    OVERLAPPED WriteOverlap;

    // constructor and destructor
    TComPort(BYTE ANumOfBuffers) throw (EPWError);
    ~TComPort(void);

    // comport read/write functions
    bool ReceiveBuffer(void) throw (EPWError);
    bool TransmitBuffer(BYTE Idx) throw (EPWError);

    bool Close(void) throw (EPWError);
    bool Open(void) throw (EPWError);

    enum TStatusFlags
    {
        COM_TRX_IDLE,
        COM_TRX_WAIT_ACK,
        COM_TRX_WAIT_REC,
        COM_TRX_WAIT_ECHO_REC,
        COM_TRX_WAIT_ECHO,
    };
};

```

## **Client code**

```

//-----

#ifndef ClientObjH
#define ClientObjH

#include <vcl.h>

class TClientObj;

// define event types
typedef void __fastcall (__closure *TNewFloatDataEvent) (TClientObj *Client, double *Data, int Num);
typedef void __fastcall (__closure *TNewIntDataEvent) (TClientObj *Client, int *Data, int Num);
typedef void __fastcall (__closure *TNewStringDataEvent) (TClientObj *Client, AnsiString String);
typedef void __fastcall (__closure *TErrorEvent) (TClientObj *Client, DWORD Flags);

//-----

class TClientObj
{
private:

    TNewFloatDataEvent FOnFloatData;
    TNewIntDataEvent FOnIntData;
    TNewStringDataEvent FOnStringData;
    TErrorEvent FOnError;

protected:

public:

    // misc
    int Type;
    AnsiString Name;

    // constructor and destructor
    TClientObj(void);
    virtual ~TClientObj(void);

```

```

// add data to model
virtual void AddData(double Data);
virtual void AddData(int Data);
virtual void AddData(double *DataArray, int Num);
virtual void AddData(int *DataArray, int Num);
virtual void AddData(AnsiString String);
virtual void AddError(DWORD Flags);

// load and save model info
virtual bool Load(TStream *InStream);
virtual bool Save(TStream *OutStream);

// add events
__property TNewFloatDataEvent OnFloatData = { read=FOnFloatData, write=FOnFloatData };
__property TNewIntDataEvent OnIntData = { read=FOnIntData, write=FOnIntData };
__property TNewStringDataEvent OnStringData = { read=FOnStringData, write=FOnStringData };
__property TErrorEvent OnError = { read=FOnError, write=FOnError };

};

//-----

#endif
//-----

#ifdef ComChannelObjH
#define ComChannelObjH

//-----

#include "..\ModCon.100\Shared\Source\MC_COMS.h"
#include "TargetObj.h"
#include "MessageLog.h"

//-----

class TComChannelObj// : public TComponent
{
private:

protected:
    static DWORD PktID;
    TMessageLog *MessageLog;
    virtual void DebugMessage(AnsiString Message);

public:

    AnsiString Name;           // Name of Channel
    AnsiString Type;           // Type of Channel
    int ID;                    // Channel Driver ID

    WORD InfoLength;

    virtual void SetMessageLog(TMessageLog *NewMessageLog);

    // constructor and destructor
    // TComChannelObj(TMessageLog *NewMessageLog, TComponent *AOwner);
    __fastcall TComChannelObj(TComponent *AOwner);
    __fastcall virtual ~TComChannelObj(void);

    // write raw data to the channel
    virtual void WriteData(int Num, void *Data) = 0;

    // data packet handling
    virtual bool AddTxPacket(void *Msg) = 0;
    virtual void* GetTxPacket(void) = 0;
    virtual bool AddInfo(void *Info) = 0;

    // clear packet queue
    virtual void ClearPending(void) = 0;

    // open and close of channel
    virtual bool Connect(void) = 0;
    virtual bool Disconnect(void) = 0;

    // load and save channel info
    virtual bool Load(TStream *InStream) = 0;
    virtual bool Load(AnsiString Regkey) = 0;
    virtual bool Save(TStream *OutStream) = 0;
    virtual bool Save(AnsiString Regkey) = 0;

};

DWORD TComChannelObj::PktID;

//-----

#endif
//-----

#ifdef FIFOH
#define FIFOH

#include "windows.h"

//-----

typedef enum
{

```



```

        FIFO_CLAMP,
        FIFO_OVERWRITE
    } TFIFOMode;

//-----

class TFIFO
{
private:
    void **Items;

    int InIdx;
    int OutIdx;
    int Size;

    TFIFOMode Mode;

    bool Full;
    bool Empty;

protected:

public:
    // constructor and destructor
    TFIFO(int FSize, TFIFOMode FMode);
    virtual ~TFIFO();

    // access operators
    void* Pop(void);
    bool Push(void *Item);
    int Count(void);
    void Clear(void);

};

//-----
#endif
//-----

#ifndef ModConH
#define ModConH

//-----

#include "TargetObj.h"

//-----

typedef struct
{
    WORD Reserved;
} TModConInfo;

typedef enum
{
    DATA_TYPE_UINT,
    DATA_TYPE_INT
} TDataTypes;

//-----

class TModCon: public TTargetObj
{
private:
    AnsiString HexConvProg;
    AnsiString HexDldProg;

    AnsiString HexFile;
    AnsiString OutFile;

    inline void SetPacketInfo(TComsCmd Cmd, TMsgFlags Flags,
        void *Pkt, BYTE *PktData);

protected:

    virtual void* GetCmdPkt(TClientObj *Client, TComsCmd CMD, TMsgFlags Flags,
        const void *Data, BYTE DataType, int Num);

public:
    // constructor and destructor
    TModCon(void);
    ~TModCon(void);

    bool SendProgram(TStream *InStream);
    bool SendProgram(AnsiString Filename);

    // send command to target
    virtual bool SendCmd(TClientObj *Client, TComsCmd Cmd, TMsgFlags Flags, const int *Data, int Num);
    virtual bool SendCmd(TClientObj *Client, TComsCmd Cmd, TMsgFlags Flags, const UINT *Data, int Num);

    // build packet and return pointer
    virtual void* GetCmdPkt(TClientObj *Client, TComsCmd Cmd, TMsgFlags Flags, const int *Data, int Num) {
        return GetCmdPkt(Client, Cmd, Flags, Data, DATA_TYPE_INT, Num); }
    virtual void* GetCmdPkt(TClientObj *Client, TComsCmd Cmd, TMsgFlags Flags, const UINT *Data, int Num) {
        return GetCmdPkt(Client, Cmd, Flags, Data, DATA_TYPE_UINT, Num); }

```

```

// return list of variables defined for target
void GetVarList(TStringList *List);

// handle message received from server
virtual void ReceivePkt(void* Buffer, DWORD Num);

// add target info and return num of bytes added
virtual void Initialize(void);

bool Load(TStream *InStream);
bool Load(AnsiString Regkey);
bool Save(TStream *OutStream);
bool Save(AnsiString Regkey);
};

//-----

#endif
//-----

#ifndef NPipeChannelClientH
#define NPipeChannelClientH

#include "syncobjs.hpp"

#include "ComChannelObj.h"
#include "NPipeChannelClientThread.h"
#include "..\PWServer\NPipeServiceThreadObj.h"
#include "FIFO.h"

//-----

typedef enum
{
    NEXT_NONE,
    NEXT_LP,
    NEXT_NP,
    NEXT_HP
} TNextPriority;

//-----

class TNPipeChannelClientThread;

class TNPipeChannelClient : public TComChannelObj
{
private:
    // queues for packets (priority based)
    TFIFO *QueueLP;
    TFIFO *QueueNP;
    TFIFO *QueueHP;

    AnsiString NPipeServer;

    TNextPriority NextAvailable;

    // thread that handles overlapped I/O with server
    TNPipeChannelClientThread *ComChannelThread;

    // critical section for FIFO access
    TCriticalSection *CriticalSection;
    // CRITICAL_SECTION CriticalSection;

protected:

public:
    bool Connected;
    bool TestAccess;

    // constructor and destructor
    __fastcall TNPipeChannelClient(TComponent *AOwner);
    __fastcall ~TNPipeChannelClient(void);

    // write raw data to the channel
    virtual void WriteData(int Num, void *Data);

    virtual void SetMessageLog(TMessageLog *NewMessageLog);

    // data packet handling
    virtual bool AddTxPacket(void *Msg);
    virtual void* GetTxPacket(void);
    virtual bool AddInfo(void *Info);

    // clear packet queue
    virtual void ClearPending(void);

    // open and close of channel
    virtual bool Connect(void);
    virtual bool Disconnect(void);

    // load and save channel info
    virtual bool Load(TStream *InStream);
    virtual bool Load(AnsiString Regkey);
    virtual bool Save(TStream *OutStream);
    virtual bool Save(AnsiString Regkey);
};

```



```

//-----
#endif
//-----

#ifndef NPipeChannelClientThreadH
#define NPipeChannelClientThreadH

//-----

#include <Classes.hpp>
#include "PError.h"
#include "TargetObj.h"
#include "NPipeChannelClient.h"
#include "MessageLog.h"

#ifdef NUM_OF_OBJ_HANDLES
#undef NUM_OF_OBJ_HANDLES
#endif

#define NUM_OF_OBJ_HANDLES      3

#define OBJIDX_PIPE_READ        0
#define OBJIDX_PIPE_WRITE      OBJIDX_PIPE_READ + 1
#define OBJIDX_PIPE_DOWRITE    OBJIDX_PIPE_WRITE + 1

//-----

class TNPipeChannelClient;

class TNPipeChannelClientThread : public TThread
{
private:
    void ConnectToPipe(void) throw (EPWError);
    void DisconnectFromPipe(void) throw (EPWError);
    void TransmitMessage(void *Msg) throw (EPWError);
    void ReadMessage(void) throw (EPWError);
    bool DispatchMessage(void);

    BYTE *WriteBuffer;
    BYTE *ReadBuffer;
    DWORD WriteBufferSize;
    DWORD ReadBufferSize;

    DWORD BytesRead;
    DWORD BytesToWrite;
    DWORD NextID;

    HANDLE ObjHandles[ NUM_OF_OBJ_HANDLES ];

    bool PendingRead;
    bool PendingWrite;
    bool Disconnecting;

    // event objects and overlapped structures used for overlapped I/O
    HANDLE ReadEvHandle;
    HANDLE WriteEvHandle;

    OVERLAPPED ReadOverlap;
    OVERLAPPED WriteOverlap;

protected:
    void __fastcall Execute(void);
    virtual void DebugMessage(AnsiString Message);
    TMessageLog *MessageLog;

    // synchronize
    BYTE *RTTReadBuffer;
    DWORD RTTNumExpected;
    TMsgFlags RTTFlags;
    TTargetObj *RTTTarget;

    void __fastcall ReturnToTarget(void);

public:
    void SetMessageLog(TMessageLog *NewMessageLog);

    TNPipeChannelClient *ComChannel;

    AnsiString PipeName;
    HANDLE Handle;

    HANDLE DoWriteEvHandle;

    bool Connected;

    bool Idle;

    __fastcall TNPipeChannelClientThread(bool CreateSuspended);
    __fastcall virtual ~TPipeChannelClientThread(void);

    bool Connect(void);
};
//-----

```

```

#endif

//-----

#ifndef PWEErrorH
#define PWEErrorH

#include <vcl.h>
#pragma hdrstop

//-----

class EPWEError
{
public:
    enum ErrorID
    {
        COM_INVALID_PORT,
        COM_BAUD_RATE,
        COM_INVALID_PORT_NUMBER,
        COM_PORT_ALREADY_OPEN,
        COM_PORT_NOT_OPEN,
        COM_OPEN,
        COM_WRITE,
        COM_READ,
        COM_CLOSE,
        COM_PURGE,
        COM_GET_STATE,
        COM_SET_STATE,
        COM_SETUP,
        COM_SET_TIMEOUTS,
        COM_CLEAR_ERROR,

        PIPE_CREATE,
        PIPE_CONNECT,
        PIPE_WRITE,
        PIPE_READ,
        PIPE_DISCONNECT,
        PIPE_SET_STATE,
        PIPE_BUSY,
        PIPE_MAX_INSTANCES,
        PIPE_CANCEL_IO,

        BUFFER_ALLOC,

        FLUSH_FILE_BUFFERS,

        EVENT_CREATE_HANDLE,
        EVENT_SET,

        WAITFORMO_FAILED,
        WAITFORMO_HANDLE_INDEX,

        CMD_INVALID_LENGTH,
        CMD_INVALID_LENGTH_FIELD,

        SERVICE_CREATE_ERROR,

        SET_SECURITY_ERROR
    };

    ErrorID Error;
    DWORD Win32Error;
    AnsiString Message;

    // constructors
    EPWEError(ErrorID AError);
    EPWEError(ErrorID AError, AnsiString AMessage);
    EPWEError(ErrorID AError, DWORD AWin32Error);
    EPWEError(ErrorID AError, AnsiString AMessage, DWORD AWin32Error);
};

//-----

#endif

//-----

#ifndef TargetObjH
#define TargetObjH

#include <grids.hpp>

//-----

typedef WORD TMsgFlags;

class TComChannelObj;

#include "..\ModCon.100\Shared\Source\MC_COMS.h"
#include "MessageLog.h"
#include "ClientObj.h"

//-----

#define MSG_FLAGS_PRIORITY_ANY    0x0000
#define MSG_FLAGS_PRIORITY_LOW   0x0001

```



```

#define MSG_FLAGS_PRIORITY_NORM      0x0002
#define MSG_FLAGS_PRIORITY_HIGH      0x0003

#define MSG_FLAGS_DISCARD_PKT        (1<<4)
#define MSG_FLAGS_RET_EXPECTED       (1<<5)
#define MSG_FLAGS_BROADCAST_GROUP    (1<<6)
#define MSG_FLAGS_BROADCAST_ALL      (1<<7)

#define MSG_FLAGS_CNL_ACK             (1<<8)
#define MSG_FLAGS_CNL_NACK           (1<<9)
#define MSG_FLAGS_TGT_ACK            (1<<10)
#define MSG_FLAGS_TGT_NACK           (1<<11)

#define MSG_FLAGS_TGT_REJECT          (1<<12)
#define MSG_FLAGS_UNKNOWN_TYPE       (1<<13)

//-----

class TTargetObj;

typedef struct
{
    DWORD      MsgSize;
    WORD       TgtInfoIdx;
    WORD       CnlInfoIdx;
    WORD       PktInfoIdx;
    TClientObj *Client;
    TTargetObj *Target;
    WORD       CTimeout;
    WORD       Flags;
} TMsgInfo;

//-----

class TTargetObj
{
private:

protected:

    AnsiString Type;

public:

    int GID;
    int DID;

    AnsiString Name;
    int ID; // Target spesific ID

    TMessageLog *MessageLog;

    TComChannelObj *ComChannel; // Comms channel
    AnsiString ComChannelName;

    WORD CmdInfoSize;
    TCmdInfo *CmdInfo;
    AnsiString CmdInfoHint;
    TStringGrid *CmdInfoGrid;

    // constructor and destructor
    TTargetObj(void);
    virtual ~TTargetObj(void);

    // download program code
    virtual bool SendProgram(TStream *InStream) = 0;
    virtual bool SendProgram(AnsiString Filename) = 0;

    // send command to target
    virtual bool SendCmd(TClientObj *Client, TComsCmd Cmd, TMsgFlags Flags, const int *Data, int Num) = 0;
    virtual bool SendCmd(TClientObj *Client, TComsCmd Cmd, TMsgFlags Flags, const UINT *Data, int Num) = 0;

    inline bool SendCmd(TComsCmd Cmd)
    { return SendCmd((TClientObj*)NULL, Cmd, MSG_FLAGS_PRIORITY_NORM, (const int*)NULL, 0); }

    inline bool SendCmd(TClientObj *Client, TComsCmd Cmd)
    { return SendCmd(Client, Cmd, MSG_FLAGS_PRIORITY_NORM, (const int*)NULL, 0); }

    inline bool SendCmd(TClientObj *Client, TComsCmd Cmd, TMsgFlags Flags)
    { return SendCmd(Client, Cmd, Flags, (const int*)NULL, 0); }

    inline bool SendCmd(TClientObj *Client, TComsCmd Cmd, TMsgFlags Flags, int Data)
    { return SendCmd(Client, Cmd, Flags, (const int*)&Data, 1); }

    inline bool SendCmd(TClientObj *Client, TComsCmd Cmd, TMsgFlags Flags, UINT Data)
    { return SendCmd(Client, Cmd, Flags, (const UINT*)&Data, 1); }

    // build packet and return pointer
    virtual void* GetCmdPkt(TClientObj *Client, TComsCmd Cmd, TMsgFlags Flags, const int *Data, int Num) = 0;
    virtual void* GetCmdPkt(TClientObj *Client, TComsCmd Cmd, TMsgFlags Flags, const UINT *Data, int Num) = 0;

    // handle message received from server
    virtual void ReceivePkt(void* Buffer, DWORD Num) = 0;

    // return list of variables defined for target
    virtual void GetVarList(TStringList *List) = 0;

    // add target info and return num of bytes added
    virtual void Initialize(void) = 0;

```

```
// load and save settings
virtual bool Load(TStream *InStream) = 0;
virtual bool Load(AnsiString Regkey) = 0;
virtual bool Save(TStream *OutStream) = 0;
virtual bool Save(AnsiString Regkey) = 0;
};

//-----
#endif
```